M11

AN INTRODUCTION TO MICROCOMPUTERS

SYBEX

VOLUME I
BASIC CONCEPTS

BY ADAM OSBORNE

SYBEX

AN INTRODUCTION TO MICROCOMPUTERS
VOLUME I — BASIC CONCEPTS

M11

AN INTRODUCTION TO MICROCOMPUTERS

VOLUME I BASIC CONCEPTS

1st Printing - 1976
2nd Printing - 1977

Published by Adam Osborne and Associates, Incorporated
P.O. Box 2036, Berkeley California 94702

For ordering and pricing information outside the U.S.A., please contact:

SYBEX (European Distributor)
313 Rue Lecourbe
F-75015 Paris
France

ARROW INTERNATIONAL (Japanese Distributor - English Translation)
No. 720, 2 Chome-4, Shiba Park
Minato-ku, Tokyo, Japan

L.A. VARAH LTD (Canadian Distributor)
2077 Alberta Street
Vancouver 10, B.C.
Canada

Taiwan Foreign Language Book Publishers Council
P.O. Box 1444
Taipei, Taiwan

TABLE OF CONTENTS

TABLE OF CONTENTS (CONTINUED)

TABLE OF CONTENTS (CONTINUED)

TABLE OF CONTENTS (CONTINUED)

LIST OF FIGURES

LIST OF TABLES

QUICK INDEX

QUICK INDEX (Continued)

QUICK INDEX (Continued)

QUICK INDEX (Continued)

QUICK INDEX (Continued)

QUICK INDEX (Continued)

QUICK INDEX (Continued)

QUICK INDEX (Continued)

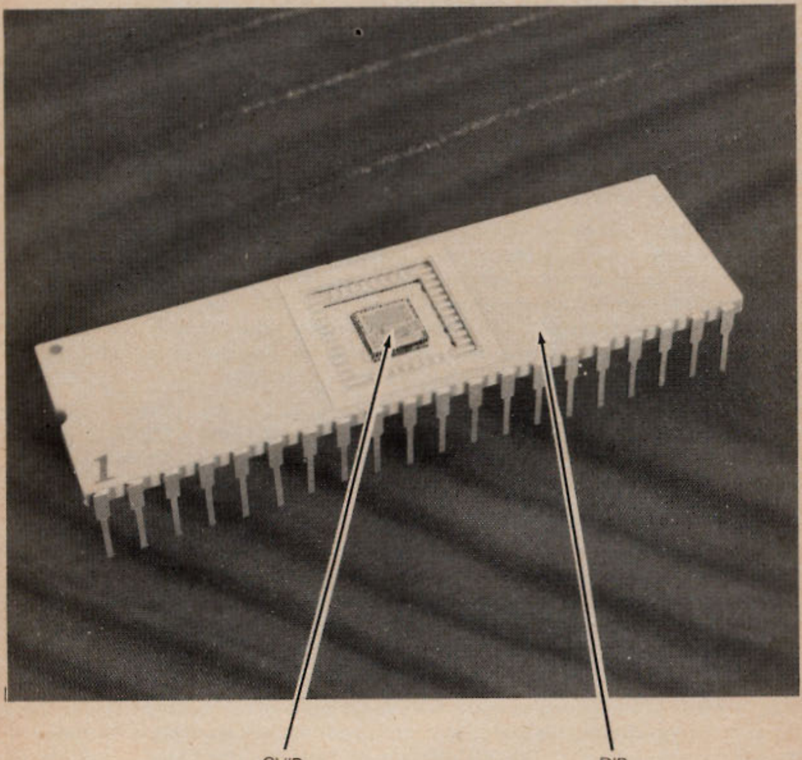


Figure 1-1. A Microcomputer Chip And DIP.

Chapter 1
WHAT IS A MICROCOMPUTER

A microcomputer is a logic device. More precisely, it is an indefinite variety of logic devices, implemented on a single chip; and because of the microcomputer, logic design will never be the same again.

The word "microprocessor" is also widely used in conjunction with microcomputers. The term "microprocessor" was coined to reflect the limited functions of these devices as compared to computers; a microprocessor, therefore, represents something less than a microcomputer. Current trends have blurred the distinction between "microprocessors" and "microcomputers"; therefore in this book we use only the term "microcomputer", identifying logic implemented on chips by specific function -- using traditional terminology.

CHIP

DEVICE

Figure 1-1 illustrates a microcomputer. The logic of the *microcomputer is on a chip, which is mounted in a Dual In-Line* Package (DIP). We refer to the DIP as a logic device, as opposed to the silicon wafer, which is a logic chip.

The microcomputer is also a digital computer, as its name would imply.

There are, indeed, striking similarities between microcomputers and other computers. The established method of comparing computers — via instruction sets, addressing modes and execution speeds — makes some microcomputers look so *similar to other computers that any distinction* between the two products appears to be a distinction in search of a difference.

But microcomputers are a new and different product, and that is why the established method of comparing computers does not apply to microcomputers. Instruction sets, addressing modes and execution speeds are of secondary importance to the microcomputer user. The distribution of logic on chips and the price of microcomputer devices are the comparisons of primary importance; and it is these comparisons that set microcomputers apart from all other types of computer, as a new and different product.

The purpose of this book is to explain not only what microcomputers are but, in addition, why they must be evaluated in a way that differs so markedly from prior computer comparisons.

The book does not assume you understand how computers work; therefore, computer concepts are described, beginning with first principles.

Microcomputers *and all other computers share a common ancestor*, however. To acquire a little perspective, we will therefore begin with a short history of computer evolution and identify the origins of the microcomputer.

THE EVOLUTION OF COMPUTERS

Today's smallest microcomputer and largest mainframe computer share a common ancestor — the UNIVAC 1 which was built out of vacuum tubes in 1950, and filled a room; yet it had less computing power than most of today's microcomputers.

UNIVAC 1, and the vacuum tube computers that followed, were used for a very limited number of "expense-is-no-object" applications, frequently to solve mathematical problems that might otherwise be impossible to solve.

The vacuum tube computer's logic was not particularly well suited to scientific applications; its logic was the immediate and natural consequence of being built out of bistable logic devices — the building block of every digital computer.

Indeed, the basic concepts for the design of a computing machine go all the way back to Charles Babbage, who in 1833 laid out the concepts that can be found, with minor variations, in every digital computer built today. In Chapters 2 and 3 we describe these basic concepts — concepts that allow computing logic to be built out of binary digits, irrespective of how the computer will be used.

What we are saying is that since the dawn of the computer industry, there have been no radical breakthroughs in the basic concepts of computing. It is advances in solid state physics that have been the computer industry's evolutionary force. New electronic technology has caused computer prices to fall so rapidly that every few years entire new markets have been engulfed by computers.

In 1960 computer prices had declined to the point where they could be used for data processing, and the day of the general purpose computer had arrived.

In 1965 the PDP-8, at $50,000, brought computers into the laboratory and the manufacturing plant's production line; and the minicomputer industry was born. Today minicomputers cost as little as $1,000, and their sphere of influence has spread as prices have come down.

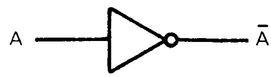
But microcomputer prices range from $5 to $250 — and we have entered an era where a computer can control a washing machine or an oven, or it can be a component in consumer products that are mass merchandised.

What are the advances in solid state physics that we speak of?

The vacuum tube is a bulky device with expensive internal elements. In the late fifties it was replaced by the transistor, a small piece of germanium metal, suitably doped with impurities.

Soon an array of discrete, low cost components were available;

A signal inverter:



An AND gate:



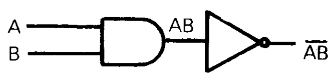
An OR gate:



An EXCLUSIVE OR gate:



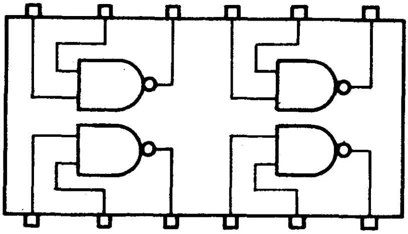
A NOT AND gate could have been:



but instead was designed as a single, new NAND gate:



Four NAND gates were built into one chip (costing the same as, or little more than, a single NAND gate) to give a quadruple 2-input positive-NAND buffer:



Devices such as the quadruple 2-input positive-NAND buffer spawned a whole range of devices, affectionately known, by a generation of logic designers, as 7400 series integrated circuits.

Indeed, the 7400 series integrated circuits, in their day, had as deep an impact on the electronics industry as microcomputers are having today; because 7400 series integrated circuits converted a generation of "circuit designers" into a generation of "logic designers" --- and the conversion occurred almost overnight.

Four gates on one chip became ten, and then a hundred, and then a thousand; today ten thousand gates worth of logic can be implemented on a single silicon chip, and the end is by no means predictable, or even in sight.

7400 INTEGRATED CIRCUITS

A chip with a number of gates on it is called an integrated circuit. If there are approximately 100 to 1000 gates on a chip, we refer to the logic as Medium Scale Integration (or MSI). At some ill-defined level, above 1000 gates of logic on a chip, we are talking of Large Scale Integration (or LSI).

MEDIUM SCALE INTEGRATION

LARGE SCALE INTEGRATION

The interesting aspect of integrated circuits is that the cost of a chip is a function of physical size — it is not a function of how much logic has been implemented on the chip. Therefore, as chips become more complex, cheaper computers can be built.

Two aspects of the amazingly shrinking computer need to be clarified:

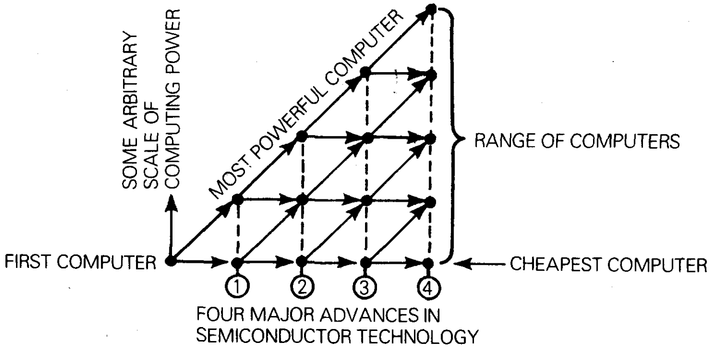
1) Does the whole computer shrink? And if not, which parts remained the same?

2) If the microcomputer is so inexpensive, why has it not eliminated all other computers?

First of all, the whole computer cannot shrink; only the electronics can. What remains is the human interface — consoles and switches, means for accepting data inputs and generating results in human readable form — all the parts of the computer that are unnecessary once a computer becomes a logic device.

The microcomputer will never eliminate all other computers because when computers are used to process data or solve scientific problems, there is a relentless economic need to make the computer more powerful. So with every major advance

...olid state electronics technology, you get two new products: a smaller yesterday's computer and a more powerful "today's" computer:



As time went by, there developed a considerable spread between the capabilities of the cheapest computer and the most powerful computer. Thus in 1965 the first arbitrary division was made — between minicomputers and large computers. We will not attempt to define what a minicomputer is, as against a large computer. A minicomputer is a minicomputer because the product's manufacturer calls it a minicomputer.

In 1970, a second arbitrary division was made, between minicomputer and microcomputer; but this time the differences between products are easier to define:

A *microcomputer is sold as one, or a very few logical devices, destined to become* components in a larger logic system.

By way of contrast, all other computers are vehicles for the execution of computer programs, each of which transiently defines the function of the computer system.

But this definitive difference between a "minicomputer" and a "microcomputer" is already blurring — for two reasons:

COMPUTER HOBBYISTS

First, the day of the computer hobbyist is here; the hobbyist builds his own computer out of a microcomputer, then writes programs for it — just like any minicomputer programmer would do.

Second, an increasing number of "microcomputers" are single chip implementations of existing "minicomputers"

THE ORIGINS OF THE MICROCOMPUTER

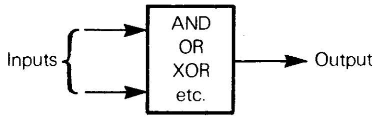
Since this is a book about microcomputers, let us look at the events which culminated in the first true microcomputer.

Datapoint Corporation of San Antonio, Texas, are a manufacturer of "intelligent terminals" and small computer systems. In 1969, they (along with Cogar and Viatron) attempted to make a "great leap forward." Datapoint engineers designed a very elementary computer, and contracted with Intel and Texas Instruments to implement the design on a single logic chip. Intel succeeded, but their product executed instructions approximately ten times as slowly as Datapoint had specified; so Datapoint declined to buy, and built their own product using existing logic components.

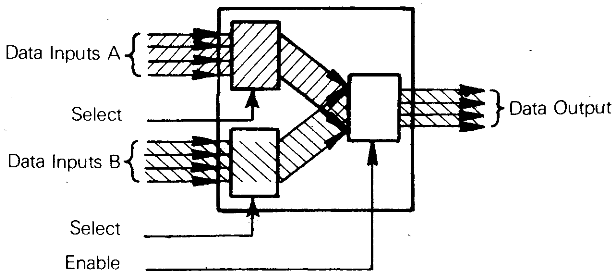
Intel were left with a computer-like logic device, whose development had been paid for. They were faced with the choice of manufacturing and selling it, or shelving it. They chose to sell it, called it the Intel 8008, and the microcomputer had arrived.

Despite the fact that the Intel 8008 was designed to perform simple data processing, the traditional job for computers, it created a market where none had existed: as a programmable *logic device*. Let us explore this concept.

In any catalog of logic components, there are perhaps ten thousand different logic devices. The simplest we have already described; simple logic gates may be illustrated as follows:

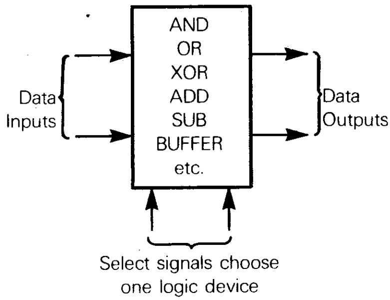


Data inputs are transformed into data outputs according to the criteria of some transfer function. But consider a more interesting logic device, a 4-bit, two-input, buffer multiplexer:

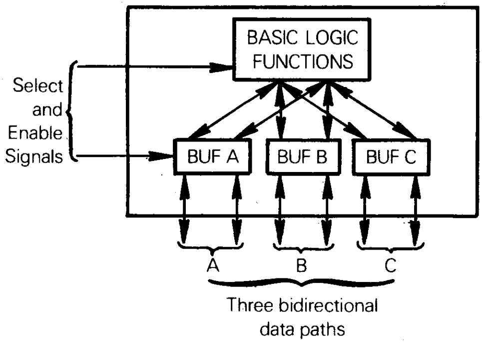


There are two interesting concepts in this buffer multiplexer. First, data are being handled in 4-bit units. Second, there are two non-data signals present: Select and Enable. Select determines which data input will become the data output. Enable determines when it will become an output.

If an LSI chip can contain thousands of gates worth of logic on it, how about condensing a catalog of logic onto a single, general purpose chip, as follows:



The general purpose chip illustrated above has a good deal of unnecessary, duplicated logic on it. Any one of the ten thousand chips listed in a catalog may be synthesized out of a few, basic logic functions — AND, OR, XOR, ADD, SUB — plus a few buffers, selects and enables:



This basic logic device can synthesize any individual logic device, or any sequence of individual logic devices.

This is the concept of the microcomputer.

ABOUT THIS BOOK

The purpose of this book is to give you a thorough understanding of what microcomputers are and how they differ from other computer products. Since the book does not assume that you have had any prior contact with computers, basic concepts are covered in considerable detail; and from basic concepts we build the necessary components of a microcomputer system.

The book does concentrate on highlighting the differences between microcomputers and minicomputers.

The book does not discuss the various technologies which are used to build logic chips because, in the end, the nature of the technology is usually quite unimportant to a user. Your application may have some key parameters such as the amount of power that you can afford to consume or the execution speeds that you can tolerate; indeed the various technologies that are used influence power consumption, execution speed and other critical factors, but where these factors are critical, selecting the right microcomputer simply involves looking at product specifications. Understanding whether the product is fabricated using N-MOS technology or C-MOS technology does not make it significantly harder or easier to understand what a microcomputer is or how to use it.

HOW THIS BOOK HAS BEEN PRINTED

Notice that text in this book has been printed in boldface type and lightface type. **This has been done to help you skip those parts of the book that cover subject matter with which you are familiar. You can be sure that lightface type only expands on information presented in the previous boldface type.** Therefore, only read boldface type until you reach a subject about which you want to know more, at which point start reading the lightface type.

Chapter 2
SOME FUNDAMENTAL CONCEPTS

The reason there is no fundamental difference between a microcomputer and any other computer is because all computer products are based on the same fundamental computing concepts — which in turn devolve to one fundamental logical concept — that of the binary digit.

BINARY DIGIT

A binary digit is a number that can have one of two values: 0 or 1. A binary digit can have no other value.

What makes the binary digit so useful is that it can be represented by any bistable device. Anything that can be "on" or "off", "high" or "low", can represent a zero in one state and a one in the other state. Figure 2-1 illustrates a bistable device. And that is all the physics you need to know in order to understand microcomputers.



Figure 2-1. A Symbolic Representation Of Binary Digits Represented By A Bistable Device

NUMBER SYSTEMS

A computer that could count no higher than one would not be a very useful machine. Fortunately, **binary digits can be used to represent numbers of any magnitude, just as a string of decimal digits can be used to represent numbers in excess of nine.** Let us therefore consider what numbers really consist of.

DECIMAL NUMBERS

When a decimal number has more than one digit, have you ever considered what each digit really represents? The two digits "11" really mean ten plus one:

$$11 = 1\times10 + 1$$

Likewise, the number 83 really means eight tens plus three:

$$83 = 8\times10 + 3$$

The number 2347 really means two thousands, plus three hundreds, plus four tens, plus seven:

$$2347 = 2\times1000 + 3\times100 + 4\times10 + 7$$

There is nothing unique or special about decimal numbers. The fact that man has ten fingers and ten toes almost certainly accounts for the universal use of base ten numbers, but any other number base would serve just as well.

BINARY NUMBERS

Because decimal digits cease to be unique with the digit 9, ten must be represented by "10", which means 1 times the number base (in this case, ten) plus 0. Using the letter "B" to represent the number base, we have:

$$10 = 1\times B + 0$$

Now in the binary numbering system, "B" does not represent ten; it represents two. Therefore, **in the binary system, 10 = decimal 2:**

$$10 = 1 \times 2 + 0$$

Similarly, in the binary system, 11 represents decimal three:

$$11 = 1 \times 2 + 1$$

Stated generally, suppose any numbering system's digits may be represented symbolically by d_i d_j, d_k, etc. If B represents the number base, then any number can be explained by this equation:

$$d_i d_j d_k d_1 = d_i \times B^3 + d_j \times B^2 + d_k \times B + d_1$$

Consider a decimal example (B=10) and binary example (B=2).

$$2\ 1\ 7\ 4 = 2 \times 10^3 + 1 \times 10^2 + 7 \times 10 + 4$$

$$d_i d_j d_k d_1 = d_i \times B^3 + d_j \times B^2 + d_k \times B + d_1$$

$$1\ 0\ 1\ 1 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2 + 1$$

CONVERTING NUMBERS FROM ONE BASE TO ANOTHER

BINARY TO DECIMAL CONVERSION

It is easy to convert numbers from one number base to another number base. Since we have only discussed decimal and binary numbers so far, consider the conversion of numbers between these two systems.

$$1\ 0\ 1\ 1 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2 + 1$$

$$2^3 = 8 \text{ and } 2^2 = 4, \text{ therefore:}$$

$$\begin{aligned} 1\ 0\ 1\ 1 &= 1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \\ &= 8 + 0 + 2 + 1 \\ &= 11 \text{ (decimal)} \end{aligned}$$

DECIMAL TO BINARY CONVERSION

Continuous division by 2, keeping track of the remainders, provides a simple method of converting a decimal number to its binary equivalent; for example, to convert decimal 11 to its binary equivalent, proceed as follows:

	Quotient		**Remainder**
$\frac{11}{2}$ =	5	+	1
$\frac{5}{2}$ =	2	+	1
$\frac{2}{2}$ =	1	+	0
$\frac{1}{2}$ =	0	+	1

1 0 1 1

Thus $11_{10} = 1011_2$

The subscripts 10 and 2 identify the numbers as base 10 and base 2, respectively.

CONVERTING FRACTIONS

The general equation to convert a fractional binary number to its decimal equivalent may be written as follows:

$$d_i d_j d_k d_1 \ldots \text{etc.} = (d_i \times B^{-1}) + (d_j \times B^{-2}) + (d_k \times B^{-3}) + (d_1 \times B^{-4}) \ldots \text{etc.}$$

where d_i d_j, d_k, etc., represents numeric digits and B represents the number base.

For example, to convert 0.1011_2 to its decimal equivalent, proceed as follows:

$$0.1011 = (1\times2^{-1}) + (0\times2^{-2}) + (1\times2^{-3}) + (1\times2^{-4})$$

where $2^{-1} = \frac{1}{2^1} = 0.5$; $2^{-2} = \frac{1}{2^2} = 0.25$; $2^{-3} = \frac{1}{2^3} = 0.125$;
$2^{-4} = \frac{1}{2^4} = 0.0625$

Thus, $0.1011_2 = 0.5_{10} + 0 + 0.125_{10} + 0.0625_{10}$
$= 0.6875_{10}$

To convert a fractional decimal number to its binary equivalent (e.g., to convert 0.6875_{10} to its binary equivalent), use the following approximation method:

0.6875	0.3750	0.7500	0.5000
× 2	× 2	× 2	× 2
(1).3750	(0).7500	(1).5000	(1).0000
↓	↓	↓	↓
1	0	1	1

Unfortunately, binary-decimal fractional conversions are not always exact; just as a fraction such as ²/₃ has no exact decimal representation, so a decimal fraction that is not the sum of 2^{-n} terms will only approximate a binary fraction.

Consider 0.42357_{10}; the binary representation of this number may be created as follows:

0.42357	0.84714	0.69428	0.38856	0.77712
× 2	× 2	× 2	× 2	× 2
(0).84714	(1).69428	(1).38856	(0).77712	(1).55424
↓	↓	↓	↓	↓
0	1	1	0	1

The answer is $0.01101\ldots_2$

As a check, let us convert back:

$$\begin{aligned} 0.01101 &= 0\times2^{-1} + 1\times2^{-2} + 1\times2^{-3} + 0\times2^{-4} + 1\times2^{-5} \\ &= 0 + 0.25 + 0.125 + 0 + 0.03125 \\ &= 0.40625_{10} \end{aligned}$$

The difference is 0.42357 − 0.40625, which equals 0.01732; this difference is caused by the neglected remainder, 0.55424. In other words, the neglected remainder (0.55424) multiplied by the smallest computed term (0.03125) gives the total error:

$$0.55424 \times 0.03125 = 0.01732$$

OTHER NUMBER SYSTEMS

Because binary numbers tend to be very long, binary digits are often grouped into sets of three or four. The numbers are now base 8 (octal) or base 16 (hexadecimal), as shown in Table 2-1. Consider the binary number:

110111101100

OCTAL NUMBERS

By grouping the binary digits into sets of three, the number is converted to octal format:

$$\underset{6}{110}\ \underset{7}{111}\ \underset{5}{101}\ \underset{4}{100} = 6754_8$$

Base 8 (octal) includes only the digits:

0, 1, 2, 3, 4, 5, 6, 7.

Decimal 8 is the same as octal 10.

HEXADECIMAL NUMBERS

By grouping the binary digits into sets of four, the number is converted to hexadecimal base:

$$\underset{D}{1101}\ \underset{E}{1110}\ \underset{C}{1100} = DEC_{16}$$

Base 16 (hexadecimal) includes the digits:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

Decimal 16 is the same as hexadecimal 10.

Table 2-1. Number Systems

HEXADECIMAL	DECIMAL	OCTAL	BINARY
0	0	0	0000
1	1	1	0001
2	2	2	0010
3	3	3	0011
4	4	4	0100
5	5	5	0101
6	6	6	0110
7	7	7	0111
8	8	10	1000
9	9	11	1001
A	10	12	1010
B	11	13	1011
C	12	14	1100
D	13	15	1101
E	14	16	1110
F	15	17	1111

BINARY ARITHMETIC

Binary numbers can be operated on in the same way as decimal numbers; in fact, binary arithmetic is much easier than decimal arithmetic. Consider binary addition, subtraction, multiplication and division.

BINARY ADDITION

The possible combinations when adding two binary digits are:

Augend	+	Addend	=	Result	+	Carry
0	+	0	=	0		
0	+	1	=	1		
1	+	0	=	1		
1	+	1	=	0	+	1

The carry, as in decimal addition, is added to the next higher binary position. For example:

This Decimal addition is equivalent to this Binary addition.

```
   3           11 ← carry
  +6           011
  --          +110
   9          ----
              1001
```

This Decimal addition is equivalent to this Binary addition.

```
  11 ← carry        1 1 ← carry
 208             11010000
 +92            +1011100
 ---            ---------
 300            100101100
```

BINARY SUBTRACTION

Microcomputers cannot subtract binary digits; they can only add. Fortunately that is no problem, since subtraction can be converted into addition.

Subtracting a decimal number is equivalent to adding the tens complement of the number.

TENS COMPLEMENT

The tens complement of a number is generated by subtracting the number from 10.

The final carry, however, must be ignored when performing decimal subtraction via tens complement addition.

Consider the decimal subtraction.

$$9 - 2 = 7$$

The tens complement of 2 is (10−2), which equals 8. The decimal subtraction can therefore be performed via the tens complement addition:

```
      9
    + 8
    ---
  = 1 7
    ↑
ignore final carry
```

Performing decimal subtraction via tens complement addition is silly, since 10−2 is no simpler to evaluate than 9−2 was. **The binary equivalent of a tens complement is a twos complement.** Performing binary subtraction via twos complement addition makes a lot of sense; moreover, twos complement logic is well suited to computers.

ONES COMPLEMENT

The twos complement of a binary number is derived by replacing 0 digits with 1 digits, and 1 digits with 0 digits, then adding 1. The first step generates a "ones complement" of a binary number. For example, the ones complement of 1011011101 is 0100100010.

Here are some other examples:

Binary number	0101
Ones complement	1010
Binary number	1010100
Ones complement	0101011

TWOS COMPLEMENT

The twos complement of a binary number is formed by adding 1 to the ones complement of that number. For example, the ones complement of 0100 is 1011:

Original number:	0100
Ones complement:	1011
	+1
Twos complement:	1100

Now look at how binary subtraction can be performed by adding the twos complement of the SUBTRAHEND to the MINUEND. First consider the following binary subtraction.

MINUEND	10001
SUBTRAHEND	−01011
DIFFERENCE	00110

The same operation can be performed by forming the twos complement of the subtrahend and adding it to the minuend. The final carry must be discarded, just as it had to be for tens complement subtraction:

MINUEND	10001
TWOS COMPLEMENT OF SUBTRAHEND	+10101
	100110

discard final carry

Thus the difference is 00110.

Consider another example:

	11001	MINUEND	11001	MINUEND
	−101	SUBTRAHEND	+11011	TWOS COMPLEMENT OF SUBTRAHEND
=	10100	=	110100	

discard final carry

When a larger number is subtracted from a smaller number, there is no carry to be discarded. Consider the decimal version of this case. 2 − 9 becomes 2 + (10 − 9), or 2 + 1. The answer, +3, is the tens complement of the correct negative result, which is −(10 − 3) = −7. Here is a binary example of the same thing:

```
   101   MINUEND          101   MINUEND
-11011   SUBTRAHEND    +00101   TWOS COMPLEMENT OF SUBTRAHEND
------                 ------
-10110   DIFFERENCE     01010   NEGATIVE ANSWER IN TWOS COMPLEMENT FORM.
```

A larger binary number has been subtracted from a smaller one. The answer on the right is negative, but it is in twos complement form; taking the twos complement of 01010 (twos complement = 10110), and assigning a minus sign, provides the same answer as on the left, −10110.

SIGN OF ANSWER IN SUBTRACTION

When performing twos complement subtraction, the final carry provides the sign of the answer. If the final carry is 1, the answer is positive. (The minuend is greater than the subtrahend.) If the final carry is 0, the answer is negative (the minuend is smaller than the subtrahend), and is in its twos complement, positive form.

BINARY MULTIPLICATION

Binary multiplication is actually easier than decimal multiplication, since each partial product, in binary, is either zero (multiplication by 0) or exactly the multiplicand (multiplication by 1). For example:

This Decimal multiplication is equivalent to this Binary multiplication:

```
  9          1001
 ×5          ×101
 --          ----
 45          1001
            0000
           1001
           ------
           101101
```

BINARY DIVISION

Binary division can be performed using the same steps as decimal division. Here is an example:

```
                     1011  ←── Quotient
Divisor ──────→ 101 |110111 ←── Dividend
                     101
                     ----
                     0011
                     0000
                     ----
                      111   ┐
                      101   │ Intermediate
                      ----  │ multiplications
                      0101  │ and subtractions
                      0101  │
                      ----  │
                         0  ┘
```

BOOLEAN ALGEBRA AND COMPUTER LOGIC

Boolean algebra is important in microcomputer applications because it provides the basis for decision making, condition testing and numerous logical operations.

Boolean algebra uses the binary digits 0 and 1 to define logical decisions. Three Boolean operators, OR, AND, and Exclusive OR (XOR) combine two binary digits to produce a single digit result. A fourth Boolean operator, NOT, complements a binary digit.

"OR" OPERATION

The OR operation is defined, for two integers I and J, by the statement:

> **If I OR J, or both, equal 1, then the result is 1. Otherwise the result is zero.**

A plus sign + is used to represent "OR". While the Boolean symbol for OR is also used to represent arithmetic addition, the two operators should not be confused; they are very similar, but they are not identical. Two binary digits are ORed as follows:

$$0 + 0 = 0$$
$$0 + 1 = 1$$
$$1 + 0 = 1$$
$$1 + 1 = 1$$

Notice that the last OR operation (1 + 1 = 1) is the only OR operation where the result differs from binary addition.

TRUTH TABLES

Logic functions are commonly defined using a Truth Table which lists the output signals associated with allowed input signal combinations. The OR Gate Truth Table is given below:

TRUTH TABLE FOR AN OR GATE

INPUTS		OUTPUT =
I	**J**	**I + J**
0	0	0
0	1	1
1	0	1
1	1	1

"AND" OPERATION

The AND operation may be defined for two integers I and J by the statement:

> **If I AND J are both 1, then the result is 1. Otherwise the result is 0.**

The dot • and $\wedge$ symbol are both used to represent the AND operation. The four possible combinations of 0 and 1 for the AND operation are:

$$0 \cdot 0 = 0$$
$$0 \cdot 1 = 0$$
$$1 \cdot 0 = 0$$
$$1 \cdot 1 = 1$$

The AND Gate Truth Table is given below:

TRUTH TABLE FOR AN AND GATE

INPUTS		OUTPUT =
I	**J**	**I • J**
0	0	0
0	1	0
1	0	0
1	1	1

"EXCLUSIVE OR" OPERATION

The EXCLUSIVE OR differentiates between input binary digits which are identical and input binary digits which are different. The output is 1 when the inputs are different and 0 when the inputs are the same. The $\oplus$ or $\forall\!\!\!\!-$ symbol is used to represent the XOR operation. The four possible combinations of 0 and 1 for the XOR operation are:

$$0 \oplus 0 = 0$$
$$0 \oplus 1 = 1$$
$$1 \oplus 0 = 1$$
$$1 \oplus 1 = 0$$

The XOR Truth Table is given below:

TRUTH TABLE FOR TWO-INPUT EXCLUSIVE OR GATE

INPUTS		OUTPUT =
A	**B**	**A $\oplus$ B**
0	0	0
0	1	1
1	0	1
1	1	0

"NOT" OPERATION

"NOT" complements any binary digit or group of digits.

NOT 1 = 0
NOT 0 = 1

Because of the nature of microcomputer logic, NOT is not a particularly significant logical operation; instead of using the NOT operation, the microcomputer's ability to generate a ones complement is employed.

Combining AND with NOT generates NAND. Combining OR with NOT generates NOR. The results of NAND and NOR are the NOT of AND and OR, respectively.

A bar is placed over a digit to represent NOT. Therefore,

$$\overline{1} = 0$$
$$\overline{0} = 1$$

COMBINING LOGICAL OPERATIONS

A microcomputer need not have all three of the Boolean operators AND, OR and Exclusive OR; some operators may be combined to generate others, as follows:

A + B is reproduced by $\overline{\overline{A} \cdot \overline{B}}$; this is illustrated as follows:

A	B	A + B	$\overline{A}$	$\overline{B}$	$\overline{A} \cdot \overline{B}$	$\overline{\overline{A} \cdot \overline{B}}$
0	0	0	1	1	1	0
0	1	1	1	0	0	1
1	0	1	0	1	0	1
1	1	1	0	0	0	1

The Exclusive OR may be generated as follows:

$$A \oplus B = (A \cdot \overline{B}) + (\overline{A} \cdot B)$$

This is illustrated as follows:

A	B	$A \oplus B$	$\overline{A}$	$\overline{B}$	$A \cdot \overline{B}$	$\overline{A} \cdot B$	$(A \cdot \overline{B}) + (\overline{A} \cdot B)$
0	0	0	1	1	0	0	0
0	1	1	1	0	0	1	1
1	0	1	0	1	1	0	1
1	1	0	0	0	0	0	0

DE MORGAN'S THEOREM

Boolean operations can be combined to produce any desired output from a set of known inputs. De Morgan's theorem is a valuable aid in designing such combinations. The theorem can be written in either of these ways:

$$\overline{A \cdot B} = \overline{A} + \overline{B}$$
$$\overline{A + B} = \overline{A} \cdot \overline{B}$$

Therefore a microcomputer only needs one Boolean operator, OR, to generate all others, since:

$$A \cdot B = \overline{\overline{A} + \overline{B}}$$

generates AND out of OR and NOT. Similarly,

$$A \oplus B = \overline{(\overline{A} + B)} + \overline{(A + \overline{B})}$$

generates XOR out of OR and NOT.

Chapter 3

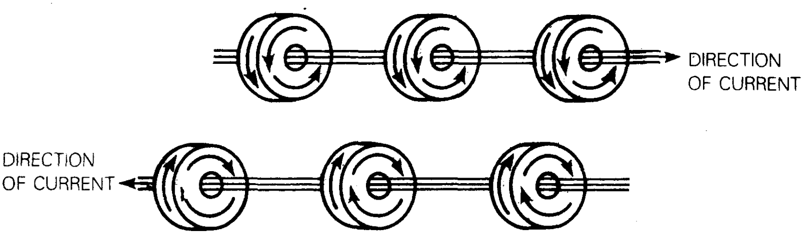
THE MAKINGS OF A MICROCOMPUTER

BITS

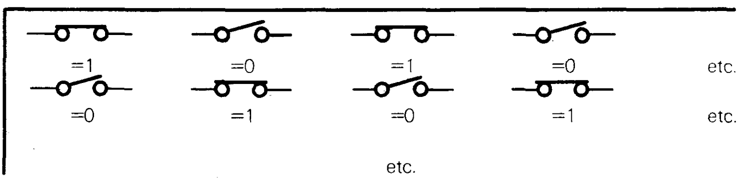
Given that BInary digiTs (referred to as BITs) are capable of being manipulated to perform any of the operations described in Chapter 2, how are these basic operations going to be harnessed in order to generate a microcomputer? First let us examine how information is stored as binary data.

MEMORY ORGANIZATION

Binary data are stored in memories. Every computer memory consists of an array of bistable elements. Minicomputers used to use and still frequently use "core" memories, which consist of minute metal "donuts" which can hold a clockwise or counterclockwise magnetic charge:



Microcomputers use semiconductor memories, which consist of an array of "gates" which may be conducting or not conducting:



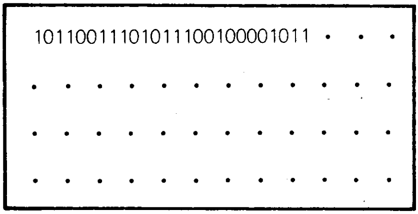
NON VOLATILE MEMORY

Core memories hold their magnetic charge even when disconnected from electric power; you can pull a core memory card out of a computer, plug it into another similar computer, and the memory data should still be intact. **Core memories are therefore said to be "non volatile".**

VOLATILE MEMORY

Semiconductor memories loose all stored data the moment you shut off their power source; therefore they are said to be "volatile".

The type of memory used with a microcomputer is unimportant. It is only necessary that the memory consist of a number of bistable, individually addressable elements, each representing a single binary digit:



There are two absolutely necessary properties which any memory must have:

1) The location where every binary digit is stored must be uniquely addressable.

2) It must be possible to read the state of every binary digit.

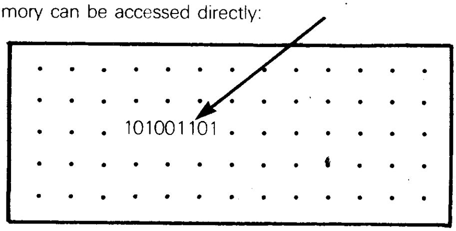
ROM

With some memories it is not possible to change the state of binary digits in the memory. If the state of binary digits can be read, but not changed, then the memory is called a Read-Only Memory, or ROM. Of course, by its very nature, any ROM memory is non volatile.

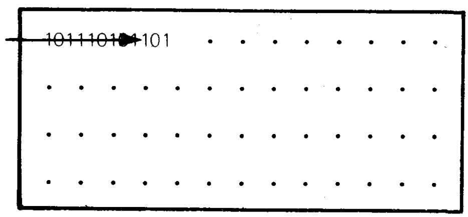
RAM

If the state of binary digits within a memory can be changed, as well as being read, then the memory is called a Read-Write memory. Read-write memories are commonly referred to as Random-Access Memories (RAM).

There is no good reason why a read-write memory, as against a read-only memory, should be referred to as a randomly accessible memory; memory is randomly accessible if individual binary digits within the memory can be accessed directly:



If binary digits within a memory were not randomly accessible, they would be sequentially accessible, which means that the tenth binary digit, for example, could only be accessed by first passing over all preceding digits:

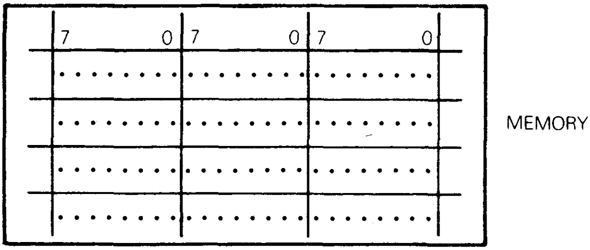


Read-only memories and read-write memories are both randomly accessible. Nevertheless, common terminology refers to read-only memories as ROMs and read-write memories as RAMs.

MEMORY WORDS

Chapter 2 explained how binary digits are combined to represent numbers in excess of 1, just as decimal digits are combined to represent numbers in excess of 9. Table 2-1 gave some binary representations of small numbers. **The primary level at which binary digits are grouped within any computer is one of the most important features of the computer and is referred to as the computer's word size.** For example, an "8-bit" computer acquires the "8-bit" label because binary data within the computer will be accessed and processed in eight binary digit units. A memory organized into 8-bit units might be visualized as follows:

WORD SIZE



Each dot in the above illustration represents a single binary digit. Each box represents an 8-bit word.

By common convention the bits of a word are numbered from right (0 for the low order bit) to left (7 for the high order bit) as illustrated above. Some computer manufacturers reverse the convention, numbering from left to right.

Table 3-1. Computer Word Sizes

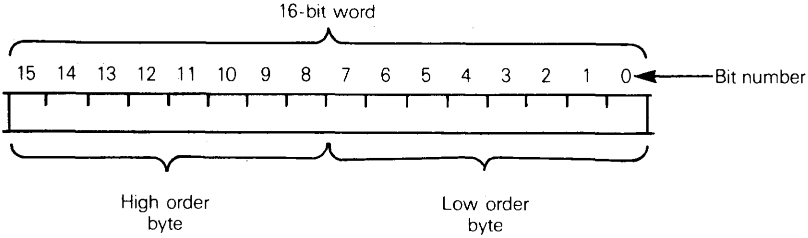
Word Size (Bits)	Microcomputers	Minicomputers	Large Computers
4	Many	None	None
6	None	A few obsolete models	None
8	Most common	A few	None
12	A few	A few	None
16	A few	Most common	A few
18	None	A few	A few
24	None	A few	A few
32	None	A few	Most common
64	None	None	Common for largest computers

A large number of different word sizes have been used by microcomputer, minicomputer and mainframe (large computer) manufacturers. Table 3-1 lists the more common word sizes and identifies those word sizes which are used by microcomputers, minicomputers and large computers.

Most microcomputers use an 8-bit word. There are a number of 4-bit microcomputers which are very much oriented toward digital logic replacement. There are also a number of 16-bit microcomputers, which tend to compete with minicomputers for their traditional markets.

THE BYTE

An 8-bit data unit is called a byte. The byte is the most universally used data unit in the computer industry; it is used even by computers that do not have an 8-bit data word. A 16-bit computer, for example, will often have memory words interpreted as two bytes:



BYTES AND WORDS

When a microcomputer has an 8-bit word size, we can refer interchangeably to "memory bytes" and "memory words"; they mean the same thing.

If a microcomputer's word size is not eight bits, then a memory word and a memory byte do not mean the same thing; a memory byte refers to an 8-bit memory unit, whereas a memory word refers to a memory unit of the microcomputer's word size.

NIBBLES

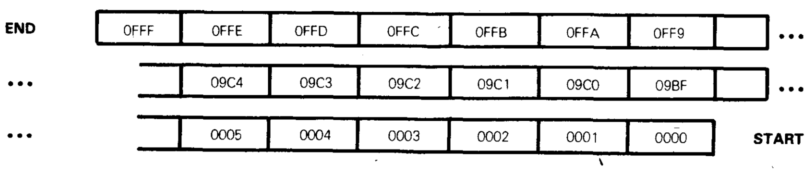
Many 4-bit microcomputers refer to the 4-bit unit as a "nibble". Thus each word of 4-bit memory is a "nibble", and two 4-bit memory words constitute a byte.

MEMORY ADDRESSES

Even though every binary digit within a memory must be uniquely addressable, binary digits are not very useful as single entities; therefore **the smallest unit of information that is usually accessed out of memory is a word.** For example, when using an 8-bit memory, each time memory is accessed, eight binary digits are referenced.

Each word of memory has a unique memory address. Words within memory have sequential memory addresses, with the first word in the memory having an address of 0, and the last word in the memory having the highest address of any word in that memory. The actual value of this highest address will depend on the size of the memory.

Thus the address of a word is its location in memory; for example, the words of a 1000_{16} (4096_{10}) word memory would be addressed and numbered as follows (in hexadecimal notation):



Conceptually, there are some subtle differences between the way minicomputer and microcomputer programmers use memories. Some of these differences are introduced now, while others are described later, since they will not be meaningful until you understand how microcomputers are used.

MINI-COMPUTER MEMORY CONCEPTS

To the minicomputer programmer, memory is simply a sequence of individually addressable RAM words, with addresses beginning at 0 and ending at some large number which depends on the size of the minicomputer's memory. It is only in rare cases that part of the minicomputer's memory will be ROM. Certainly **a minicomputer programmer never needs to worry about the physical implementation of memory.** So long as data can be stored and retrieved on demand, where and how this happens is irrelevant.

MICRO-COMPUTER MEMORY CONCEPTS

The microcomputer programmer will be very interested in how memory is implemented, because there are many applications where a microcomputer-based product, once developed, will be sold in tens of thousands of units. This being the case, it is very important that the number of discrete components within the microcomputer system be kept to a minimum, since every extra (and therefore unnecessary) component will be multiplied by tens of thousands — thus increasing costs.

The microcomputer programmer has a further interest in memory organization because almost all microcomputer based products use ROM for some part of memory. The reason ROM is desirable when using microcomputers is that ROM is safe. Since no binary digit within a ROM can be modified, nothing stored in a ROM memory can be accidentally erased. This is a very desirable characteristic in a product which may end up in obscure or inaccessible locations.

The microcomputer user thinks of memory as semiconductor chips. Figure 3-1 illustrates a 1024-bit memory device in a dual in-line package.

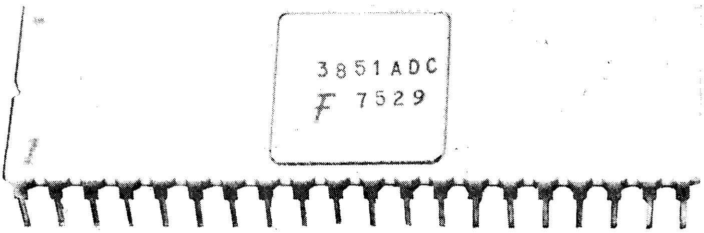
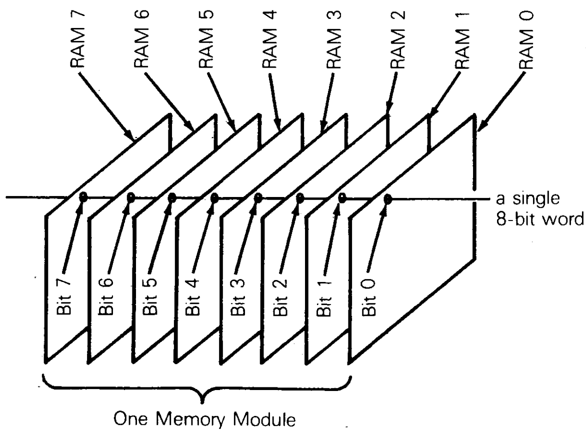


Figure 3-1. A 1024-bit Memory Device.

Usually ROM memory is implemented in single chips. For example, a microcomputer may have 1024 8-bit words of ROM memory on a single chip. This single chip will have a capacity of 8192 binary digits, divided into (and accessed as) 1024 8-bit units. A microcomputer programmer will be interested in how memory is implemented, because moving out of the memory space provided by a single ROM device requires an additional ROM device to be added to the system.

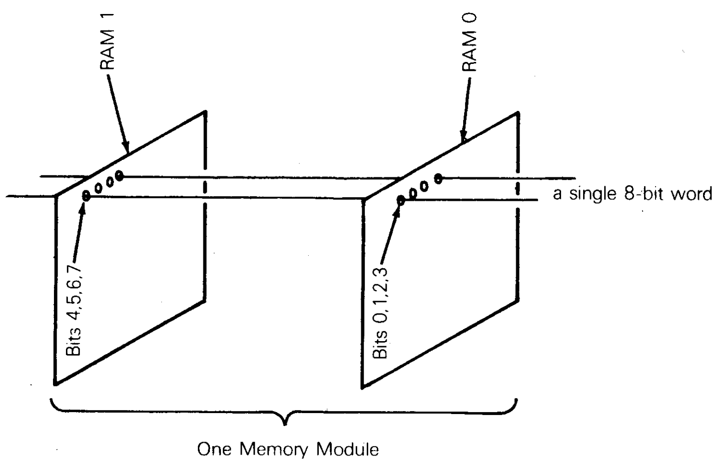
Read-write memory requires more logic than read-only memory, since the individual bits of a read-write memory can be changed as well as being read. Therefore, read-write memory is commonly implemented on more than one chip. In a very simple case, eight RAM chips may implement 8-bit read-write memory words, with each chip contributing one bit of the word:



We will refer to this set of eight chips as a memory module.

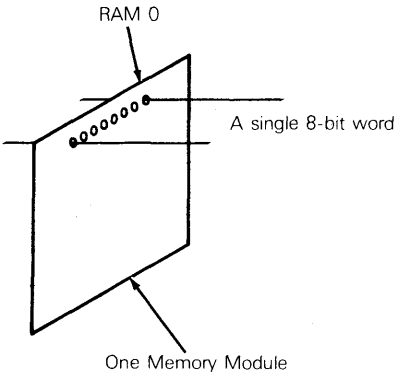
MEMORY MODULE

Small microcomputer systems may use fewer memory chips to implement small read-write memories. For example, two RAM chips may each contribute four bits of an 8-bit word:



Now there are two memory chips in the memory module.

RAM memory is also available like ROM, with entire words implemented on a single chip:



RAM memories cost increases, in terms of cents-per-bit, when fewer chips are used to implement a single memory word. Thus implementing an 8-bit read-write memory word using eight RAM chips generates the cheapest memory. Having the entire 8-bit word on one RAM chip generates the most expensive memory.

RAM CHIP MEMORY SIZE

Currently 4096-bit RAM chips are common, and 16,384-bit RAM chips are almost here. 65,536-bit RAM chips should be available in commercial quantities by late 1977 — for between $5 and $10 per chip.

DYNAMIC RAM

STATIC RAM

There are two types of RAM memory: dynamic RAM and static RAM. Dynamic RAM, which is cheaper, can only hold data for a few milliseconds; therefore dynamic RAM must constantly be refreshed by having the contents of memory words rewritten. Dynamic RAM refresh is handled automatically by some microcomputer systems; other microcomputer systems require external refresh logic when you use dynamic RAM. Static RAM costs more, but once data have been written into it, the data will stay there as long as power is being input.

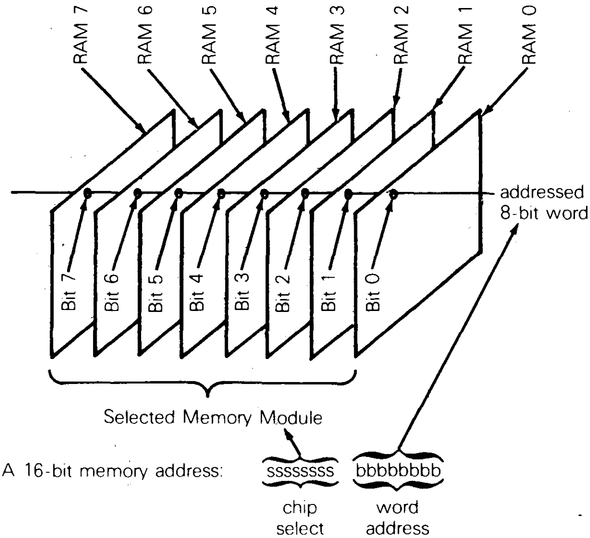
Once again, as a microcomputer user, you will be very interested in knowing exactly which memory addresses have been assigned to read-write memory. This is because memory addresses translate into RAM chips. An extra byte of data memory may require eight new RAM chips, which, multiplied by 10,000, is expensive.

Because, as a microcomputer user, you will be constantly concerned with how memory chips relate to memory addresses, memory addresses will take on a significance that differs markedly from the world of mini and larger computers. Specifically, **every memory address may be visualized as consisting of chip select bits and word address bits.**

The chip select bits select one or more chips that constitute a memory module.

The word address bits identify one memory word within the selected memory module.

Suppose 8-bit memory words are implemented on eight separate memory chips. The chip select bits will select an eight-chip memory module. The word address bits will identify one memory word, as follows:



SIZE OF MEMORY ADDRESS

The number of word address bits required by a memory module will depend on chip size. For example, if a chip contains part or all of 256_{10} memory words, then the word address will consist of eight binary digits:

Smallest word address = 00000000 = 00_{16} = 00_{10}

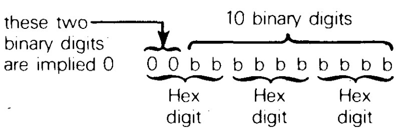
Largest word address = 11111111 = FF_{16} = 255_{10}

A larger memory chip may have part or all of 1024_{10} memory words; then the word address will consist of ten binary digits:

Smallest word address = 0000000000 = 000_{16} = 000_{10}

Largest word address = 1111111111 = $3FF_{16}$ = 1023_{10}

Notice that ten binary digits create three hexadecimal digits as follows:



SIZE OF CHIP SELECT

The number of chip select bits will be a function of the microcomputer's architecture; but concatenating the number of chip select bits with the number of word address bits generates the microcomputer's maximum memory capacity. For example, if the microcomputer can address $65{,}536_{10}$ ($FFFF_{16}$) memory words, 16 binary digits will be required to express the largest allowed memory address:

```
1111111111111111
└┬─┘└┬─┘└┬─┘└┬─┘
 F    F    F    F
```

Now if 1024-word memory chips are used, the word address consumes ten binary digits, which leaves six binary digits for the chip select; in other words, maximum memory will consist of 64 memory modules, with 1024_{10} words per module, and the 16-bit memory address must be interpreted as follows:

```
        16 binary digit memory address
┌──────────────────────┴──────────────────────┐
s s s s s s b b b b b b b b b b
└────┬────┘ └──────────┬──────────┘
6 binary         10 binary digit word
digit chip       address, holds values
select,          from 0 to 3FF16
selects one      (1023_10)
of 40_16
(=64_10)
memory
modules
```

The important thing to remember is that the microcomputer sets the TOTAL number of memory address binary digits; how they are split between chip select and word address depends on the type of memory chips used — it is entirely up to the logic designer.

Look again at how a total word address is created in a real case. If, as illustrated earlier, the word address within the chip is 0110101110_2 ($1AE_{16}$) and the chip select is 000111_2 (07_{16}), then a 16 binary digit word address is created as follows:

```
                      chip select      word address
                     ┌─────┴─────┐ ┌──────────┴──────────┐
Total word address = 0 0 0 1 1 1 0 1 1 0 1 0 1 1 1 0_2
                     └──┬──┘ └──┬──┘ └──┬──┘ └──┬──┘
                        1       D       A       E_16
```

There is no reason why available memory addresses need to be continuous, or even need to start at zero. For example, **a microcomputer system may include one ROM chip implementing 8-bit words and a RAM module made up of two RAM chips, each implementing four bits of an 8-bit word.**

If the ROM chip has a chip select of 000001_2 and a capacity of 1024_{10} memory bytes, then allowed ROM memory addresses will be 0400_{16} through $07FF_{16}$:

```
                      chip select     word address
First ROM address  =  0 0 0 0 0 1  0 0 0 0 0 0 0 0 0 0₂
                        0        4        0        0₁₆
```

```
                      chip select     word address
Last ROM address   =  0 0 0 0 0 1  1 1 1 1 1 1 1 1 1 1₂
                        0        7        F        F₁₆
```

ADDRESS SPACE

Observe that 1024_{10} bytes of memory will now have addresses 1024 through 2047_{10} or 0400_{16} through $07FF_{16}$. **We refer to this range of memory addresses as the memory module's address space. If the RAM module has a select of 000110_2 and each chip holds 256 x 4 bits,** then the two RAM chips constitute a memory module, and provide 8-bit RAM memory words with addresses 1800_{16} through $18FF_{16}$.

```
                      chip select     word address
First RAM address  =  0 0 0 1 1 0  0 0 0 0 0 0 0 0 0 0₂
                        1        8        0        0₁₆
```

```
                      chip select     word address
Last RAM address   =  0 0 0 1 1 0  0 0 1 1 1 1 1 1 1 1₂
                        1        8        F        F₁₆
```

Addresses in the range 1800_{16} through $18FF_{16}$ constitute the RAM module's address space.

INTERPRETING THE CONTENTS OF MEMORY WORDS

A memory word consists of a number of binary digits; therefore, binary digits are the only form in which information can be stored in a word of memory.

An 8-bit memory word can contain 256 (2^8) different patterns of 0's and 1's. The pattern of zeros and ones within a memory word may be interpreted in any one of the following ways:

1) **Pure binary numeric data that stand alone.**
2) **Binary numeric data that must be interpreted as one part of a multiword data unit.**

3) **A data code; that is, a bit pattern subject to some arbitrary predefined set of interpretations.**

4) **An instruction code; that is, a bit pattern which is to be transmitted to the microcomputer. The microcomputer will decode the bit pattern and interpret it as an identification of those operations which the microcomputer logic must immediately perform.**

This is the only important concept to understand at this time:

Upon examining the contents of any word of memory, it is impossible to determine whether the memory word contains numeric data, a code, or an instruction.

In Chapter 4 you will learn how a microcomputer takes care of the fact that the contents of any memory word may be interpreted in a number of different ways. But first we will describe each interpretation of a memory word.

STAND ALONE, PURE BINARY DATA

Consider first pure binary data, subject to no special interpretations.

It is important to understand that you can represent pure binary data, on paper, as a binary number, an octal number or a hexadecimal number; the choice is purely a question of reader convenience and has no impact whatsoever on the data word. Here is an example for an 8-bit data word:

```
      4       E  <-- Hexadecimal
   0 1 0 0 1 1 1 0 <-- Binary
    1   1     6  <-- Octal
```

$$01001110_2 = 116_8 = 4E_{16}$$

Here is an example for a 16-bit data word:

```
     D       B       8       B  <-- Hexadecimal
  1 1 0 1 1 0 1 1 1 0 0 0 1 0 1 1 <-- Binary
  1   5     5     6     1     3  <-- Octal
```

$$1101101110001011_2 = 155613_8 = DB8B_{16}$$

The choice of binary, octal or hexadecimal representation for a memory word's contents is not data interpretation; it is merely an alternative way of writing the same thing on a piece of paper.

INTERPRETED BINARY DATA

The contents of a memory word, interpreted as pure binary data, may stand alone, or may be part of a larger numeric unit. For example, an 8-bit memory word standing alone can represent numeric values ranging from 0 to 255_{10}. A 16-bit memory word, on the other hand, can represent numeric values ranging from 0 to $65{,}535_{10}$. **There is no reason why 8-bit memory words should not be interpreted in pairs.** Now the contents of each 8-bit memory word will be interpreted as the lower-half or the upper-half of a 16 binary digit unit:

MULTIWORD BINARY DATA

```
                 One 16-bit word
0 1 0 0 1 0 1 1   1 0 0 1 1 1 1 0

One 8-bit word.       One 8-bit word.
Upper half of a       Lower half of a
16-binary digit       16-binary digit
number.               number.
```

There is, in fact, no limit to the number of memory words that may be concatenated to generate very large numbers. Here is an example of a 48-bit number:

```
                    One 48-bit word
100111011101111010100110001110110101101100101101
 byte 5   byte 4   byte 3   byte 2   byte 1   byte 0
```

Suppose six 8-bit memory words are required to represent a single numeric unit, as illustrated above. Normally, these six memory words would be contiguous; that is, they would have memory addresses adjacent to each other. However, **there is nothing in the logic of a microcomputer that demands the many bytes of a multibyte number be contiguous; contiguous multibyte numbers are easier to process; that makes contiguous organization desirable.**

MULTIBYTE BINARY ADDITION

What about multibyte arithmetic? Numbers that occupy many memory words can be added, subtracted, multiplied or divided using the rules described in Chapter 2. Nothing was said in Chapter 2 about the number of binary digits associated with any number. Thus if a 16-bit number is stored in two adjacent 8-bit memory words, binary addition rules described in Chapter 2 would still be used, but the 16-bit numbers would have to be added in two steps as follows:

```
                                      word 1      word 0

                                     10011101    10000110
                                    +00101010    11010100
carries from word 0 and word 1 ----> 0           1
                                    =11001000    01011010

                                      step 2      step 1
```

The carry, if any, from each step is added to the low order digit of the next step.

The logical extension of the above example to numbers stored in three, four or more memory words is self-evident. **Here, for example, is how two numbers, each occupying four 8-bit words, would be added in four steps:**

	word 3	word 2	word 1	word 0
	10110100	10000101	01101011	11011010
+	01111010	10111010	01000010	00111001
1	1	0	1	
=	00101111	00111111	10101110	00010011
	step 4	step 3	step 2	step 1

MULTIBYTE BINARY SUBTRACTION

There is a catch to subtracting multibyte numbers. Recall from Chapter 2 that binary data are subtracted by taking the twos complement of the subtrahend and adding it to the minuend. Consider two 16-bit numbers stored in 16-bit memory words. **The logic associated with subtracting one 16-bit number from another is very straightforward and may be illustrated as follows:**

$$23A6_{16} - 124A_{16} = 115C_{16}$$

$23A6_{16}$	=	0010001110100110
Twos complement of $124A_{16}$	=	1110110110110101
		1
Answer	=	0001000101011100
		1 1 5 C

Now consider the same subtraction where the numbers are stored in two adjacent 8-bit memory words. The subtraction may be directly reproduced as follows:

		word 1	word 0	
23_{16}	=	00100011	10100110	= $A6_{16}$
ones complement of 12_{16}	=	11101101	10110101	= twos complement of $4A_{16}$
			1	
		00010001	01011100	
		1 1	5 C	
		step 2	step 1	

Notice that only the low order byte of the number is twos complemented. The high order byte is ones complemented. While this may seem confusing at first, it really is not. If you visualise a multibyte number as a single numeric unit, then it is self-evident that when the twos complement of the multibyte number is generated, 1 will be added to the low order byte of the multibyte number only:

Twos complement of $124A_{16}$	=	1110110110110101
		1
		1110110110110110

Twos complement of $4A_{16}$ → 10110101
Ones complement of 12_{16} → 11101101

		11101101	10110101
			1
Twos complement of $124A_{16}$	=	11101101	10110110

SIGNED BINARY NUMBERS

A microcomputer that could only process positive numbers would not be very useful. **What about negative numbers? Here, for the first time, we get into the question of interpreting binary coded data. A very effective industry convention interprets the high order bit of a number as a sign bit. If this bit is 1, the number is negative. If this bit is 0, the number is positive:**

0bbbbbbb represents a positive, 7-bit number

1bbbbbbb represents a negative, 7-bit number

Table 3-2 gives the interpretations for 8-bit signed binary data. Observe that negative numbers are coded as the twos complement of their positive counterparts. Here are some examples:

$$+02_{16} = 00000010 \qquad -02_{16} = \begin{array}{r} 11111101 \\ 1 \\ \hline 11111110 \end{array}$$

$$+6A_{16} = 01101010 \qquad -6A_{16} = \begin{array}{r} 10010101 \\ 1 \\ \hline 10010110 \end{array}$$

When eight binary digits are being interpreted as a signed number, the range of numbers is from -128_{10} to $+127_{10}$. When sixteen binary digits are interpreted as a signed binary number, numbers must fall in the range -32768_{10} to $+32767_{10}$.

Table 3-2. Signed Binary Numeric Interpretations

BINARY	EQUIVALENT DECIMAL	HEXADECIMAL
10000000	−128	80
10000001	−127	81
10000010	−126	82
10000011	−125	83
⋮	⋮	⋮
11111110	−2	FE
11111111	−1	FF
00000000	0	0
00000001	1	1
00000010	2	2
00000011	3	3
⋮	⋮	⋮
01111101	+125	7D
01111110	+126	7E
01111111	+127	7F

The beauty of using this scheme to represent signed numbers is that it calls for no special logic when performing arithmetic operations. Providing an arithmetic operation does not generate an answer which is too large to fit in the available space, you can ignore the sign of a number until you wish to interpret an answer; at that time, examining the high order bit of the answer indicates whether it is positive or negative. If the high order bit of the answer is 1, then the answer is negative and by taking the twos complement of the answer, the pure binary, positive representation of the answer is created. Here are some examples:

63_{16} → 01100011
$-3A_{16}$ → 11000101 ones complement of $3A_{16}$
1
$=29_{16}$ 00101001
2 9 Answer = $+29_{16}$
This 0 indicates a positive result

$3A_{16}$ 00111010
-63_{16} 10011100 ones complement of 63_{16}
1
$=29_{16}$ 11010111

This 1 indicates a negative result.
Take the twos complement: 00101000
1
00101001
2 9 Answer = -29_{16}

Now consider the above example, rewritten as $(3A_{16}) + (-63_{16}) = (-29_{16})$. (-63_{16}) will be represented by the twos complement of $+63_{16}$.

$+63_{16}$ = 01100011
ones complement of $+63_{16}$ = 10011100
-63_{16} = 10011101

Therefore:

$3A_{16}$ 00111010
$+(-63_{16})$ 10011101
$=(-29_{16})$ 11010111

This 1 indicates a negative result

00101000 Taking the twos complement of the answer generates the positive representation
1
00101001
2 9

Observe that using twos complement notation to represent signed binary numbers, $3A_{16} - 63_{16}$ and $3A_{16} + (-63_{16})$ have identical binary representations, which is only to be expected of a viable scheme for representing negative numbers.

MULTIWORD SIGNED BINARY NUMBERS

Multiword signed binary numbers generate no special problems so long as you understand that operations must be performed one word at a time. This is illustrated below for the simple case of 16-bit, signed binary data, which generates the same results when handled as single 16-bit words or as two 8-bit words.

```
                         one 16-bit word
sign bit ——> 0 1 1 0 0 1 0 1 1 0 0 0 0 1 0
            8-bit word 1      8-bit word 0
```

Consider the subtraction of two 16-bit, signed binary numbers, where each number is stored in two 8-bit words. As for unsigned multiword addition, signed multiword addition proceeds in two steps, as follows:

		word 1	word 0
$1A2C_{16}$	=	00011010	00101100
$+(-0810_{16})$	=	11110111	11110000
$= 121C_{16}$	=	00010010	00011100
		1 2	1 C
sign bit		step 2	step 1

Observe that -0810_{16} is generated by taking the twos complement of 0810_{16} as follows:

0810_{16} = 0000100000010000

ones complement = 1111011111101111

twos complement = 1111011111110000

BINARY CODED DECIMAL

It is possible to code decimal numbers using binary digits. Four binary digits can represent values from 0 to F_{16}, or from 0 to 15_{10}. By ignoring binary digit combinations above 9, decimal numbers can be coded, two digits per 8-bit memory word, or four digits per 16-bit memory word. **Table 3-3 identifies the combinations of four binary digits that may be interpreted as decimal numbers.** When binary digits are being used to represent decimal numbers, the result is called Binary-Coded Decimal (BCD) data.

NEGATIVE BCD DATA

Signed binary number rules cannot be applied to BCD data, since BCD demands that binary data be interpreted in 4-bit units:

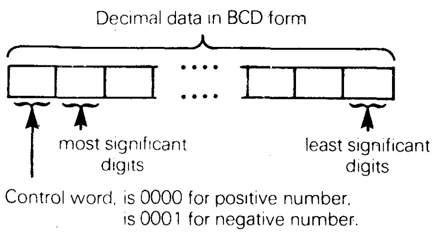
```
etc. . . .  n n n n  n n n n  n n n n  n n n n
             digit    digit    digit    digit
               3        2        1        0
```

Each 4-bit digit can have one of the bit patterns shown under the BCD column of Table 3-3. An 8-bit word uses the high order bit for all numbers in excess of 79_{10}. If the high order bit is needed to represent decimal digits 8 or 9, then it cannot be used to represent a sign.

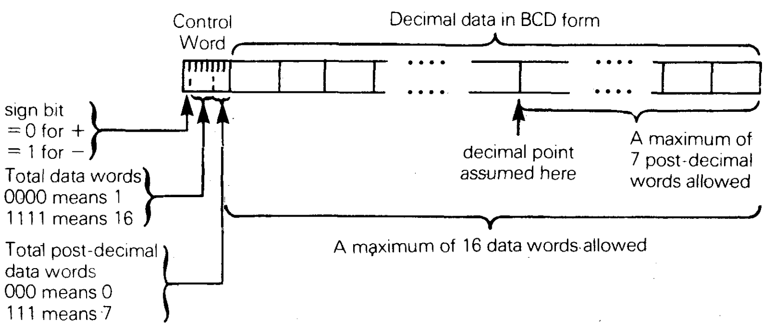
Table 3-3. Binary Representation of Decimal Digits

BINARY	HEXADECIMAL	BCD
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	Illegal
1011	B	Illegal
1100	C	Illegal
1101	D	Illegal
1110	E	Illegal
1111	F	Illegal

The sign of signed BCD numbers is therefore represented using a special "control" word which must precede the first data word of a multiword BCD number. There are no common rules for control word format, but here is a simple example and a complex example. First the simple example:



Now the complex example:



BCD data cannot be added and subtracted using straightforward binary addition and subtraction rules. Here are some examples of the errors that could result:

Decimal		BCD	Decimal		BCD
23	=	00100011	54	=	01010100
+47	=	01000111	−26	=	11011010
=70		01101010	=28		00101110
		6 Illegal			2 Illegal

Note that 11011010 is the twos complement of the binary representation of 26.

BCD ARITHMETIC

In order to perform BCD arithmetic, special rules must be applied, and the carry out of each BCD digit must be recorded. Consider the addition:

```
  97
 +68
=165
```

It is insufficient to record the fact that there was a carry out of the high order digit addition. Any intermediate carry out of the low order digit addition must also be recorded. Here are some examples showing the status of the carry (C) and the intermediate carry (IC):

C IC	C IC	C IC	C IC
↓ ↓	↓ ↓	↓ ↓	↓ ↓
0 0	0 1	1 0	1 1
21	29	91	97
+32	+32	+32	+68
=53	=61	=123	=165

Conceptually, BCD addition is performed as follows:

1) Add the two numbers.
2) Serially add 6 to each invalid digit, starting with the low order digit. (This "jumps over" the six invalid bit combinations to yield the correct BCD representation.)

For example, $25_{10}+19_{10}$ =

```
         0010  0101
         0001  1001
         ----------
         0011  1110
add 6:   0000  0110
         ----------
         0100  0100  = 44₁₀
```

This method is rather clumsy to computerize. The following process, as described for two digits of a BCD number, is more efficient; it adds 6 to each digit, then based on the carry statuses, subtracts those sixes which were not needed:

1) Using binary addition, add 66_{16} to the first number (the augend).
2) Add the second number (the addend) to the sum generated in step 1. The carry generated in this step reflects the true carry to the next higher digit.

3) Using binary addition, add a factor to the sum from step 2. The factor to be added depends on the carry (C) and intermediate carry (IC) statuses as follows:

C	IC	Factor
0	0	$9A_{16}$
0	1	$A0_{16}$
1	0	FA_{16}
1	1	00_{16}

Here are some addition examples:

	23 +32 =55	29 +34 =63	92 +32 =124	87 +79 =166
Augend =	00100011	00101001	10010010	10000111
66_{16} =	01100110	01100110	01100110	01100110
Step 1 sum =	10001001	10001111	11111000	11101101
Addend =	00110010	00110100	00110010	01111001
Step 2 sum =	10111011	11000011	00101010	01100110
C/IC =	0 0	0 1	1 0	1 1
Factor =	10011010	10100000	11111010	00000000
Step 2 sum =	10111011	11000011	00101010	01100110
Answer =	01010101	01100011	00100100	01100110
=	5 5	6 3	2 4	6 6
C (from Step 2) =	0	0	1	1

BCD subtraction is performed for two digits of a BCD number via these two steps:

1) Add the twos complement of the subtrahend (the number being subtracted) to the minuend (the number being subtracted from). The carry generated in this step reflects the true carry to the next higher digit.

   Recall that when multiword numbers are subtracted, only the lowest order word of the subtrahend is twos complemented. Higher order words are ones complemented.

2) Perform step 3 as described for BCD addition.

Here are some subtraction examples:

	75 −21 +54	71 −28 +43	25 −71 −46	21 −78 −57
Subtrahend =	00100001	00101000	01110001	01111000
Twos compl. =	11011111	11011000	10001111	10001000
Minuend =	01110101	01110001	00100101	00100001
Step 1 sum =	01010100	01001001	10110100	10101001
C/IC =	1 1	1 0	0 1	0 0
Factor =	00000000	11111010	10100000	10011010
	01010100	01001001	10110100	10101001
Answer =	01010100	01000011	01010100	01000011
=	5 4	4 3	5 4	4 3
C (after AC) =	1	1	0	0

When performing BCD subtraction, a negative result is indicated by a final carry of 0 (as for binary subtraction), but in keeping with the decimal representation of numbers, the numeric value of the negative answer is in tens complement form, not in twos complement form. Thus the answer to 25 − 71 appears as 54, which is 100 − 46, and the final carry is 0. Similarly, the answer to 21 − 78 appears as 43, which is 100 − 57, and the final carry is 0.

CHARACTER CODES

A computer would not be very useful if it required data to be entered as a sequence of binary digits, or if answers were output in one of the uncoded or coded binary formats. It must be possible for a computer to handle text and other nonnumeric information.

If we bear in mind that the combination of binary digits within any memory word can be re-used in any number of ways, then all the binary codes which have been used to represent numeric data, as described so far, can all be re-used to represent letters of the alphabet, digit characters, or any other special printed characters.

So long as a program correctly interprets the binary digits of a memory word, then confusion and ambiguities cannot arise.

For example, if you as the programmer decide to use memory words with addresses $0A20_{16}$ through $0A2A_{16}$ to hold binary-coded decimal data, then it is up to you, the programmer, in your subsequent logic, to remember that the binary data in these memory words must be interpreted as binary-coded decimal digits — and any other interpretation will cause errors.

Likewise, if memory words $12A4_{16}$ through $12A6_{16}$ are reserved to hold binary data which are to be interpreted as character codes, then the fact that character codes have exactly the same binary digit pattern as binary-coded decimal words is irrelevant. So long as program logic correctly interprets the contents of memory words, errors cannot arise; and if program logic does not correctly interpret the contents of memory words, then program logic is in error and must be corrected.

In order to handle text, a complete and adequate set of necessary characters includes:

CHARACTER SETS

26 lower case letters
26 upper case letters
approximately 25 special characters (e.g. [+/@!#, etc.)
10 numeric digits

The above character set adds up to 87 characters. A six binary digit group allows 64 combinations of 0 and 1 binary digits (2^6), which is insufficient to represent 87 characters. A seven binary digit group allows 128 possible arrangements of 0 and 1 binary digits, which is sufficient for our needs.

The 8-bit byte has been universally accepted as the data unit for representing character codes. The two most common character codes, listed in Appendix A, are known as the American Standard Code for Information Interchange (ASCII) and Extended Binary Coded Decimal Interchange Code (EBCDIC). ASCII is used by all minicomputer and microcomputer manufacturers.

PARITY

Eight binary digits are used to represent characters where seven binary digits suffice. This being the case, the eighth binary digit is frequently used to test for errors, and is referred to as a parity bit; it is set to 1 or to 0, so that the number of 1 bits in the byte is either always odd or always even.

If odd parity is selected, then the parity bit will be set or reset so that the total number of 1 bits is always odd. Here are some examples:

Parity Bit ↓

10000000	Number of 1 bits = 1
00000001	Number of 1 bits = 1
11001011	Number of 1 bits = 5
11011111	Number of 1 bits = 7
01010100	Number of 1 bits = 3

If even parity is selected, then the parity bit is set or reset so that the total number of 1 bits will always be even. Here are some examples:

Parity Bit ↓

00000000	Number of 1 bits = 0
10000001	Number of 1 bits = 2
01010101	Number of 1 bits = 4
10010101	Number of 1 bits = 4
11111111	Number of 1 bits = 8

The parity bit is used to ensure that between the creation of a character byte and reading it back, no bit was erroneously changed. If, for example, parity is odd, then whenever an even number of 1 bits is detected in a character byte, clearly the byte must be in error. Similarly, if even parity is selected, then whenever an odd number of bits are detected in a character byte, the byte must be in error.

Here is an example of how a message might be stored in a sequence of contiguous memory words using ASCII character codes with even parity:

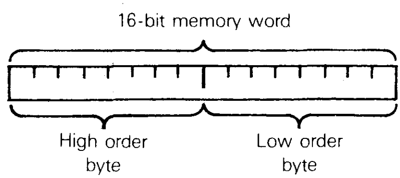
E	n	t	e	r	blank	
11000101	11101110	01110100	01100101	01110010	00100000	etc.

A few comments regarding parity and error codes would be useful at this point.

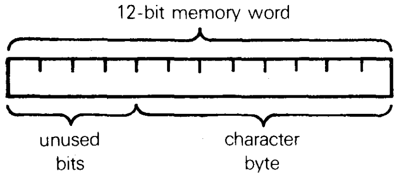
Clearly, **the high order bit of a byte can be used as a parity bit only when the byte contents are being interpreted as character codes.** If the contents of the byte are being interpreted as any form of binary data (coded or uncoded), then the high order bit has already been specified as an integral part of the byte's data contents; therefore, this bit cannot be used as a parity bit, so coded and uncoded binary data cannot have parity checked.

Many elaborate schemes are used, not only to check that sequences of binary digits contain no errors, but further to detect what these errors are and to make appropriate corrections. These error correction codes have nothing in particular to do with minicomputer or microcomputer concepts; therefore, they are not discussed in this book.

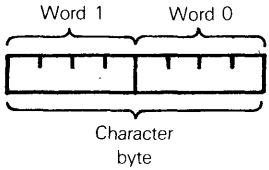
Microcomputers that have word sizes other than eight bits still use the byte to represent character codes. A 16-bit microcomputer will pack two bytes into each memory word as follows:



A microcomputer that uses 12-bit words will store the character code in the lower eight of the 12 bits and will waste the higher four bits as follows:



A 4-bit microcomputer creates a byte out of two contiguous memory words:



4-bit microcomputers may actually be better suited to BCD applications such as hand held calculators. These applications treat 4-bit data units as unique entities — one BCD digit per 4-bit data unit; the 8-bit byte is not significant, so the fact that the 4-bit microcomputer always handles data in 4-bit units can greatly simplify programming.

INSTRUCTION CODES

Memory words have so far been interpreted as data of one form or another. The contents of a memory word can also be interpreted as a code, identifying an operation which is required of the microcomputer.

Consider the simple example of binary addition. Assume that the contents of the memory word with address 0A30 is to be added to the contents of the memory word with address 0A31, and the sum is to be stored in the memory word with address 0A31. Program steps to accomplish this binary addition may proceed as follows:

1) Identify the address of the first memory word to be added.

2) Transfer the contents of this memory word to the microcomputer.

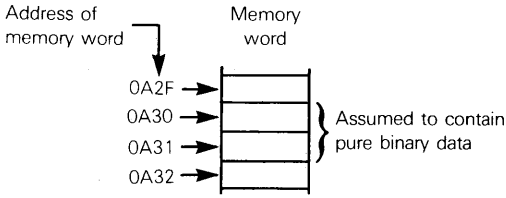
3) Identify the address of the second memory word to be added.

4) Add the contents of this memory word to the memory word which was transferred to the microcomputer in step 2.

5) Identify the address of the memory word where the sum is to be stored.

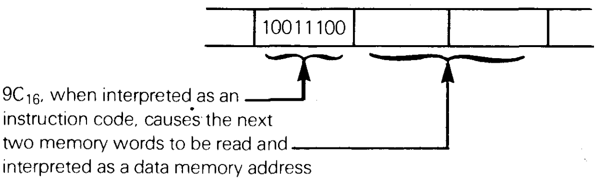
6) Transfer the sum to this memory word.

Program logic specifies that the memory words with addresses 0A30 and 0A31 are to contain pure binary data:

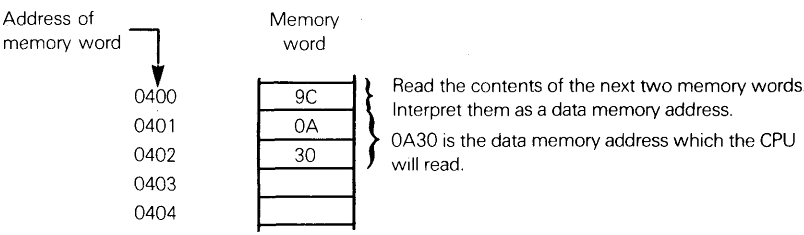


Let us assume that the six program steps are to be stored in memory words with addresses starting at 0400; we will create some instruction codes to implement the six-step binary addition.

The instruction code which identifies memory addresses will occupy three bytes, as follows:



We can now start to create a program as follows:



Step 2 requires the contents of the addressed memory word to be read and interpreted as data. Note that there is no need for the instruction to specify what kind of data the memory word contains. You, the programmer, must remember what kind of data the addressed word contains, and not try to do anything incompatible with the data type. So far as the microcomputer is concerned, data is pure binary data — nothing more, nothing less.

Let us assume that the binary code:

01000000

if interpreted as an instruction, causes the contents of the addressed data memory word to be read and interpreted as data. Our program now becomes:

Address of memory word	Memory word	
0400	9C	
0401	0A	Step 1
0402	30	
0403	40	Read contents of addressed data word (0A30 from step 1) and interpret as pure binary data
0404		

Step 3 is a repeat of step 1, only a different data memory address ($0A31_{16}$) needs to be specified. Our program now becomes:

Address of memory word	Memory word	
0400	9C	
0401	0A	Step 1
0402	30	
0403	40	Step 2
0404	9C	
0405	0A	Read 0A31, as a data memory address into the CPU
0406	31	
0407		

Step 4 is a variation of step 2; however, instead of simply reading the contents of the addressed data memory word, the data memory word is added, using binary addition, to the data memory word that was previously read; assume that this operation is identified by the instruction code:

10000000

Our program now increases one step as follows:

Address of memory word	Memory word	
0400	9C	
0401	0A	Step 1
0402	30	
0403	40	Step 2
0404	9C	
0405	0A	Step 3
0406	31	
0407	80	Add contents of addressed data memory word to data word currently stored in the CPU
0408		
0409		

Step 5 is a repeat of step 3; the address of the data memory word where the sum is to be stored, $0A31_{16}$ is the data memory word most recently addressed (in step 3), so the instruction need not be repeated for step 5. Instead we will proceed to step 6, and assume that the binary word:

01100000

when interpreted as an instruction code, causes data to be output to the most recently addressed data memory word. The complete program now looks like this:

Address of memory word	Memory word	
0400	9C	
0401	0A	Step 1
0402	30	
0403	40	Step 2
0404	9C	
0405	0A	Step 3
0406	31	
0407	80	Step 4
0408	60	Store data word in CPU in memory word addressed by data memory address
0409		

Providing the microcomputer can be told in some way that a sequence of instruction codes will be found beginning at memory word 0400_{16}, then the fact that each of these memory words may have a pattern of 1 and 0 binary digits which is also valid binary data, or ASCII characters, is irrelevant.

Notice that the program creates memory addresses which identify memory words assumed to contain pure binary data. It is assumed that you, as a microcomputer programmer, will make sure that these memory words do indeed contain binary data. If, by mistake, instructions elsewhere in your program store character codes in these same memory words, then upon being commanded to do so, the microcomputer will add two character codes as though they were binary data. Only the strange results which are created will alert you to the fact that a mistake has been made.

Illustrating the concept of a microcomputer program via this six-step binary addition is, of course, just a beginning.

How does the microcomputer perform the operations required by the instruction code? That question will be answered in Chapter 4.

What does the microcomputer demand of external logic in order to complete the operations specified by an instruction code? That question will be answered in Chapter 5.

How do you write a microcomputer program? We will address that question in Chapter 6.

Chapter 4

THE MICROCOMPUTER CENTRAL PROCESSING UNIT

The logic of a microcomputer is implemented on one or more chips. Chips are packaged in DIPs, as was illustrated in Figure 1-1. Most microcomputer DIPs have 40 pins, but other pin counts, ranging from 28 to 64 may be found.

CPU

MICROPROCESSOR

The one thing you can say for sure is that **one of your DIPs will contain logic that is referred to as a Central Processing Unit, or "CPU"; this is the DIP commonly called a "Microprocessor".** Whatever else a microcomputer has, or lacks, it must have a CPU.

The logic that constitutes a CPU can differ wildly from one microcomputer to the next; underlying these variations there are certain necessities, however. Our purpose in this chapter is to identify these necessities.

Recall from Chapter 3 that the contents of a memory word may be interpreted in one of the following ways:

1) As pure binary data
2) As coded binary data
3) As a character code
4) As an instruction code.

These four ways of interpreting the contents of a memory word can be broadly separated into two major categories: data and instructions.

Pure binary data, coded binary data, and character codes have one thing in common: they are all data. The contents of data memory words can be operated on, or can be used in conjunction with the contents of other data words.

Instruction codes are input to the CPU as a means of identifying the next operation which you want the CPU to perform. A sequence of instruction codes, stored in memory, constitute a program.

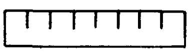
Consider the six-step binary addition program described at the end of Chapter 3. Let us examine, in the following paragraphs, the logic that the CPU must contain to perform this binary addition.

CPU REGISTERS

ACCUMULATOR

The CPU must have one or more registers in which data that have been fetched from memory can be stored; we will call these registers Accumulators. Since the majority of microcomputers use an 8-bit word size, that is the word size we will adopt — and assume an 8-bit Accumulator:

7 6 5 4 3 2 1 0 Bit number

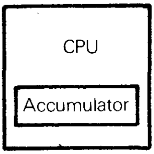


Accumulator (A)

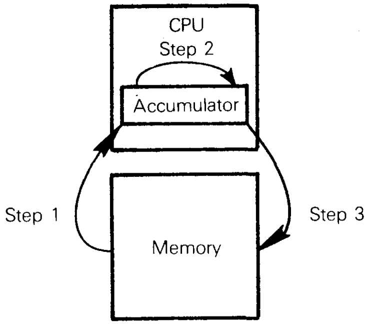
To keep things simple, we will, for the moment, assume that there is just one Accumulator in the CPU.

The data fetched in step 2 of the binary addition program described in Chapter 3 is stored in the Accumulator.

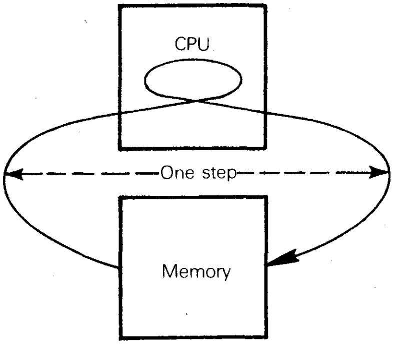
The CPU usually operates on the contents of an Accumulator, rather than accessing memory words directly. Does this make sense? Remember, the Accumulator is a register within the logic of the CPU:



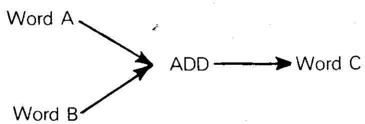
Since data is permanently stored in external memory, you may argue that operating on data in a CPU register forces programs to define a three step instruction sequence:



where one step would do:



Unfortunately the "One step" illustrated above does not always work. Some CPU operations require two memory words' contents to be fetched from memory:



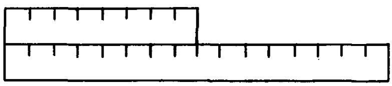
Sometimes the CPU operates on a single word:

Word A → Ones Complement → Word C

Or we can look at the One step operation negatively: must we always spend time fetching data from memory, then returning results to memory? The answer is no. Every memory access takes time and logic. By having a few data storage registers in the CPU, we can have one memory access for every five CPU operations (approximately); and that is better than two memory accesses for every CPU operation, which the one step sequence requires. Therefore nearly all microcomputers have Accumulators, or Accumulator type registers in the CPU.

DATA COUNTER

In order to access a data memory word, either to read its contents or to store data into the memory word, the data memory word address must be identified; this address is held in a register which we will call the Data Counter. The size of the Data Counter will depend on the maximum amount of memory that the microcomputer can address. Here is a 16-bit Data Counter, which can address up to 65,536 words of data memory:



A microcomputer's CPU can have one or more Data Counters. To keep things simple, we will, for the moment, assume that the CPU has only one Data Counter.

Referring again to the binary addition program described in Chapter 3, the data memory addresses $0A30_{16}$ and $0A31_{16}$ would be held in the Data Counter register.

In order to access a word of data memory, the CPU needs an Accumulator to store the contents of the accessed data word, and a Data Counter to store the address of the data word being accessed.

Similarly, in order to handle instruction codes, the CPU is going to need a register to store instruction codes, and a register to store the address of the memory word from which the instruction code is going to be fetched.

INSTRUCTION REGISTER

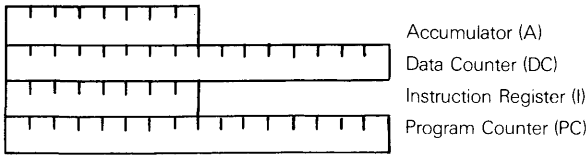
The instruction code is stored in an Instruction register; the CPU will always interpret the contents of the Instruction register as an instruction code.

PROGRAM COUNTER

The address of the memory word from which the instruction code will be fetched is provided by a register which we will call the Program Counter.

The Program Counter is analogous to the Data Counter, but the Data Counter is assumed to always address a data memory word, while the Program Counter is assumed to always address a

program memory word. We now have the following four registers:



There is one important conceptual difference between the Data Counter and the Program Counter. **By storing instruction codes in sequential memory words, the problem of creating instruction code addresses in the Program Counter is resolved.** All that is needed is to find some way of loading an initial address into the Program Counter. If, after accessing a memory word to fetch an instruction code, the contents of the Program Counter is incremented by 1, then the Program Counter will be left pointing to the memory word containing the next instruction code.

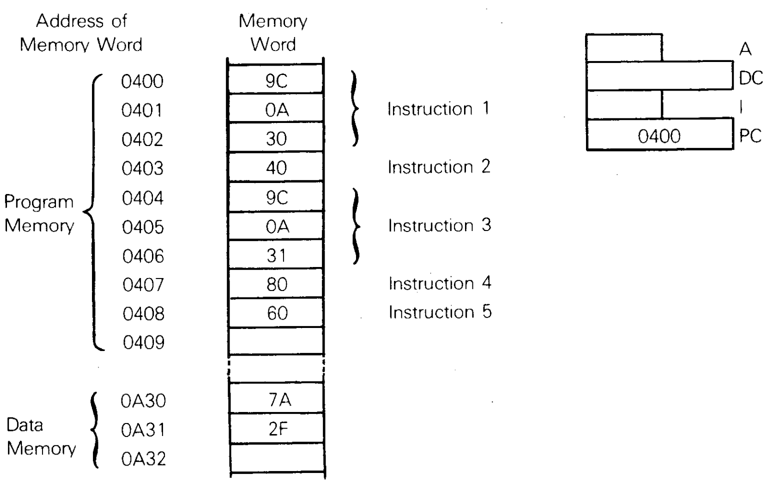
The Data Counter, on the other hand, is not likely to have long runs of sequential memory accesses. Only when data are stored in multiword units, or data tables are held in contiguous memory words, will the Data Counter be required to access sequential memory locations. Even when the Data Counter is required to access sequential memory locations, it is not clear whether the Data Counter should start at a low memory address and increment, or start at a high memory address and decrement. Therefore, **CPU logic is going to have to provide the microcomputer user with a great flexibility when it comes to setting addresses in the Data Counter.**

HOW CPU REGISTERS ARE USED

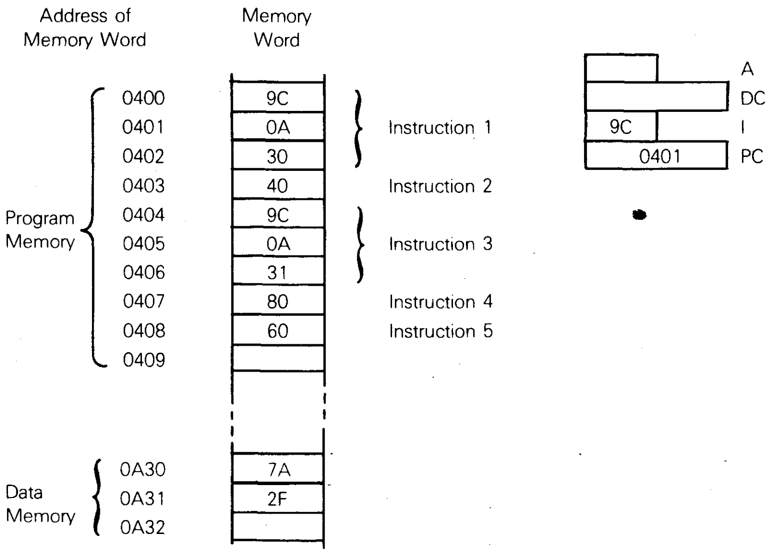
In order to fully understand how microcomputer CPU registers are used, we will step through the binary addition program of Chapter 3, showing how the contents of the four registers change. We will, from here on, refer to each step of the program as an Instruction, since in reality, each step, as illustrated, merely identifies an instruction's binary code.

INSTRUCTIONS

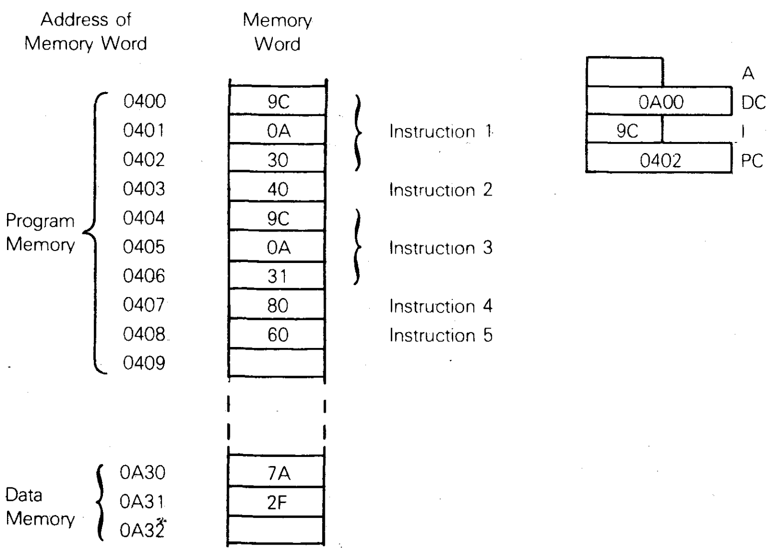
Initially the Program Counter (PC) contains 0400_{16}, the address of the first instruction word in program memory; the contents of other registers is unknown. We assume, to complete the illustration, that data memory words $0A30_{16}$ and $0A31_{16}$ initially contain $7A_{16}$ and $2F_{16}$, respectively.



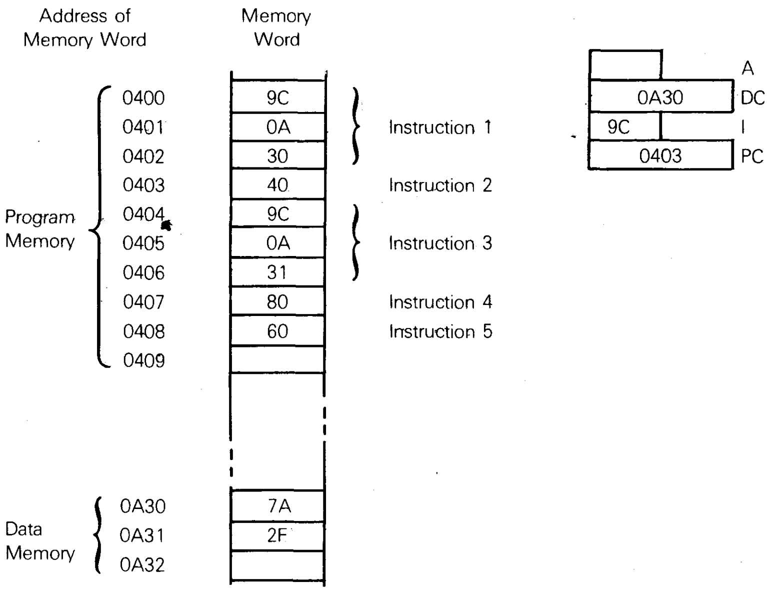
The CPU loads the contents of the memory word addressed by PC into the Instruction register (I), thus ensuring that the memory word contents will be interpreted as an instruction code. The CPU then increments the contents of PC:



The code 9C, appearing in the Instruction register, causes CPU logic to implement two steps. First, the contents of the memory word addressed by PC is fetched from memory, but is stored in the high order byte of Data Counter (DC). The CPU then increments the contents of PC:



Next, the contents of the memory word addressed by PC is fetched from memory and stored in the low order byte of DC. Again the CPU increments the contents of PC:

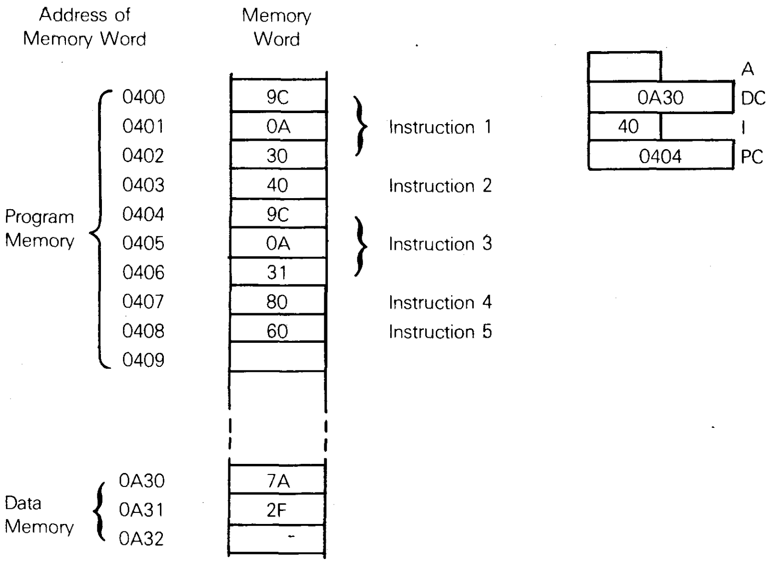


LITERAL, OR IMMEDIATE DATA

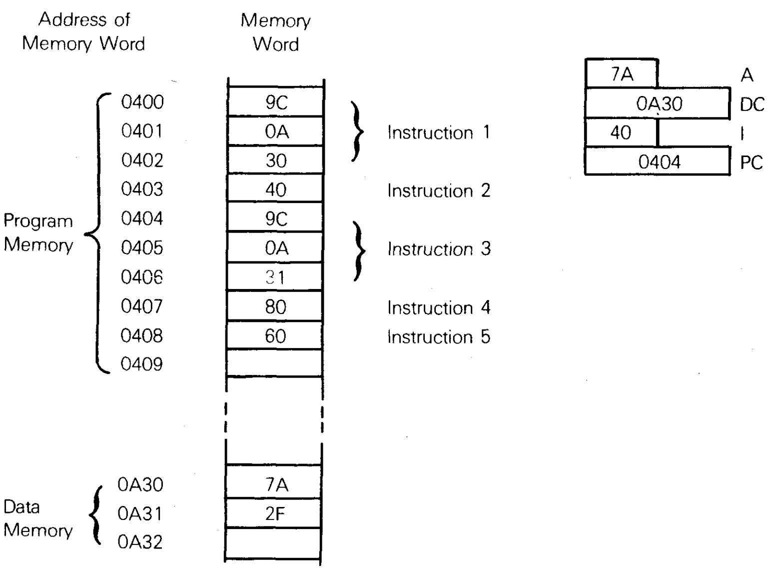
Execution of Instruction 1 is now complete. Observe that the contents of memory words 0401_{16} and 0402_{16} have been loaded into the DC register, even though these two memory words are in program memory, and are addressed by the Program Counter (PC). The important concept here is that the instruction code requires data to follow it immediately. Non-instruction codes, required by and appearing immediately after instruction codes in program memory, are called Literal, or Immediate data.

For example, in Instruction 1, memory words 0401_{16} and 0402_{16} contain the immediate data $0A30_{16}$. The instruction code 9C, fetched from memory word 0400_{16}, identifies the way in which the immediate data $0A30_{16}$ must be interpreted by the CPU.

Let us now continue to Instruction 2. Upon completion of Instruction 1, the CPU fetches the contents of the memory word addressed by PC, then increments PC. Having been given no other specific instructions, the contents of the fetched word are stored in the I register, to be interpreted as an instruction code:



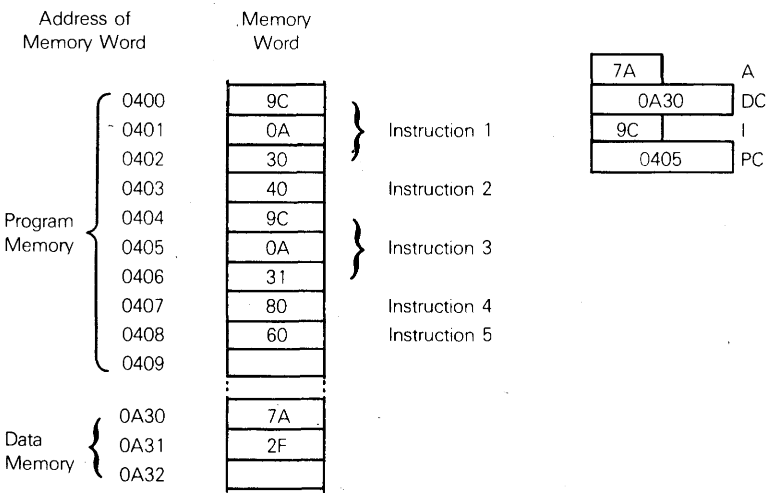
This instruction code causes the CPU to fetch the contents of the memory word addressed by DC and to load this memory word into the Accumulator (A):



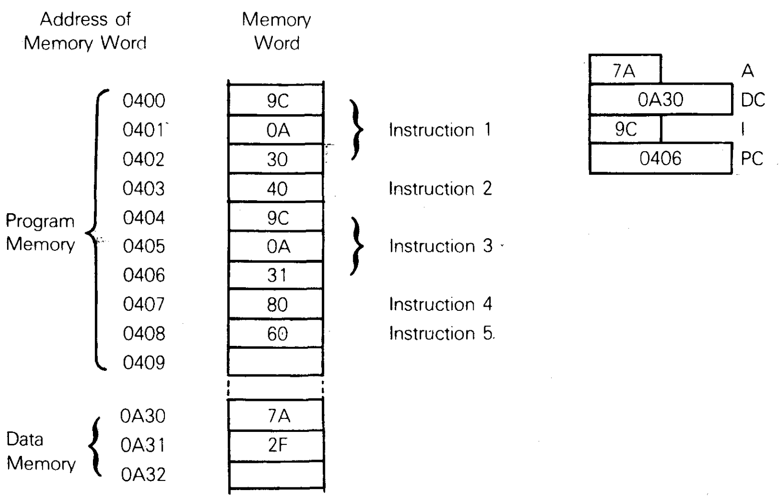
Notice that neither DC contents nor PC contents are incremented. PC contents are not incremented because 7A is not immediate data; it was fetched from data memory. DC contents are not incremented since there is no guarantee that data words will, in the normal course of events, be referenced sequentially.

Instruction 2 has now completed execution, and PC addresses the next program memory word, which contains the instruction code for Instruction 3.

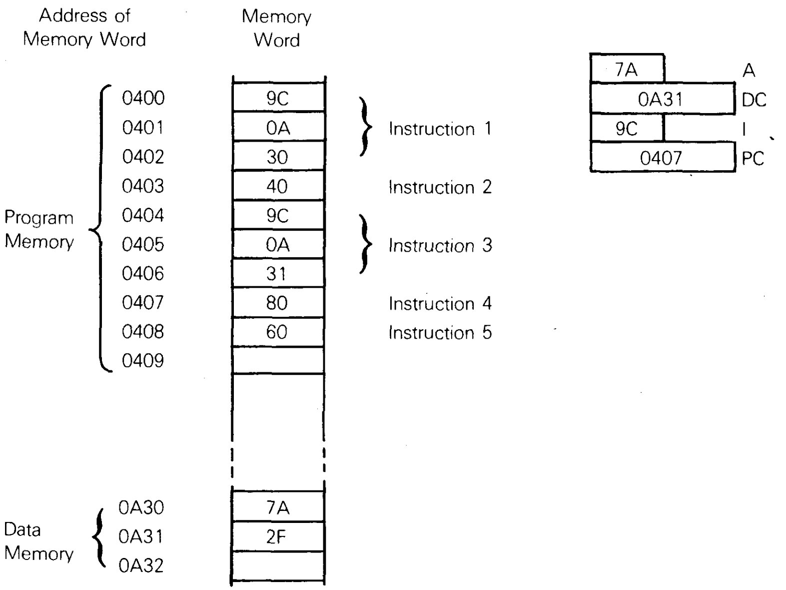
Instruction 3 is a repeat of Instruction 1, except that the literal data $0A30_{16}$ has been replaced by $0A31_{16}$. As for Instruction 1, CPU registers undergo changes in three steps when Instruction 3 executes; step 1 fetches the instruction code to the I register:



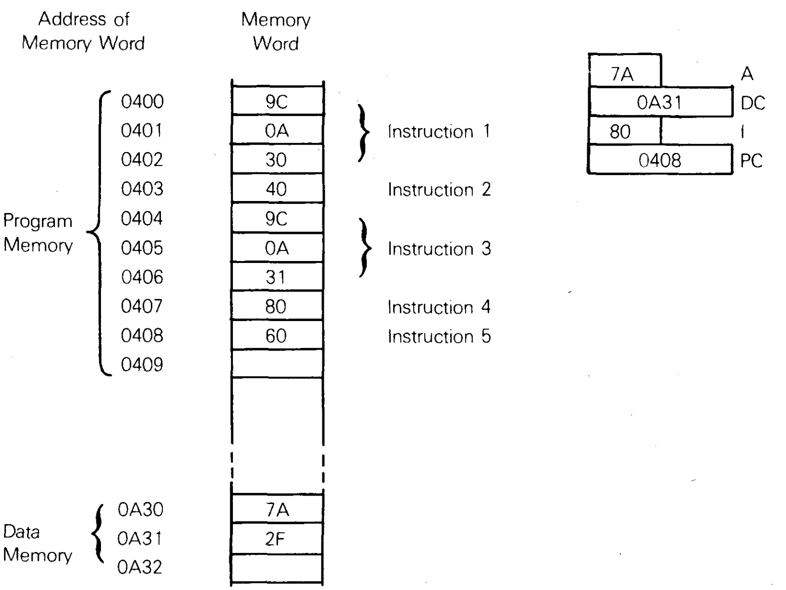
Step 2 fetches 0A from word 0405_{16} and stores it in the high order byte of the DC register; by chance, that is what the DC register contained, so no change appears in the DC register:



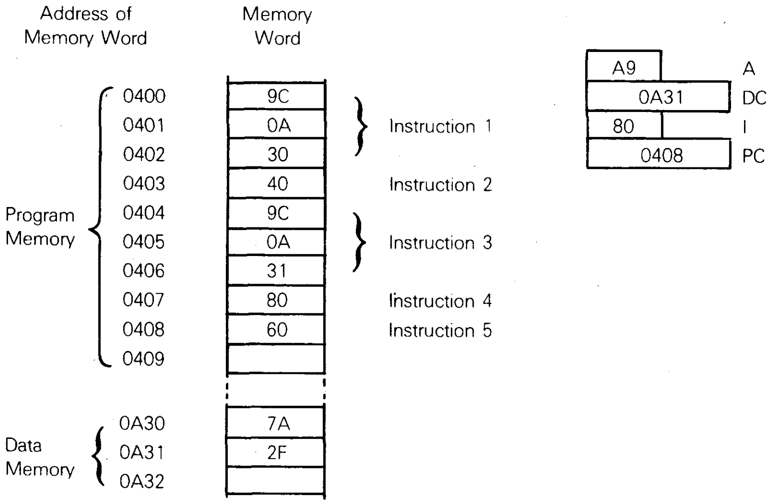
Step 3 changes the low order byte of DC:



Instruction 3 has now completed execution, and execution of Instruction 4 is ready to begin. As with the previous instructions, the CPU automatically starts by loading the contents of the memory word addressed by PC into I:



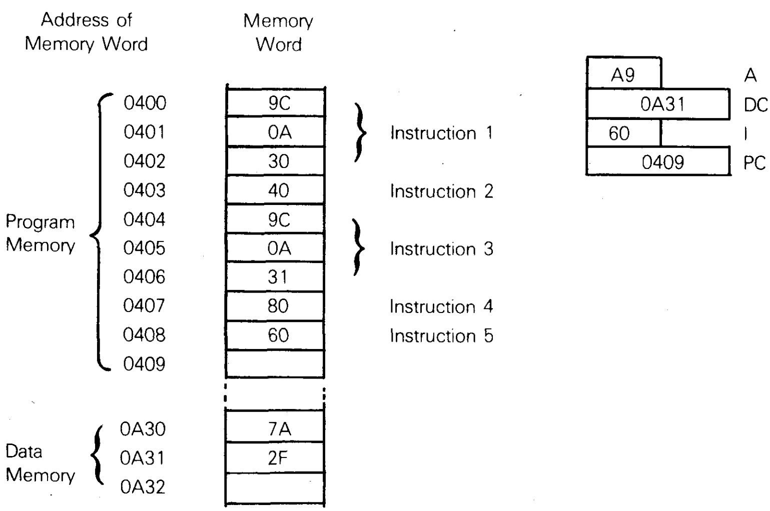
The instruction code 80 requires the CPU to fetch the contents of the data word addressed by DC and to add this data word to the contents of the Accumulator (A):



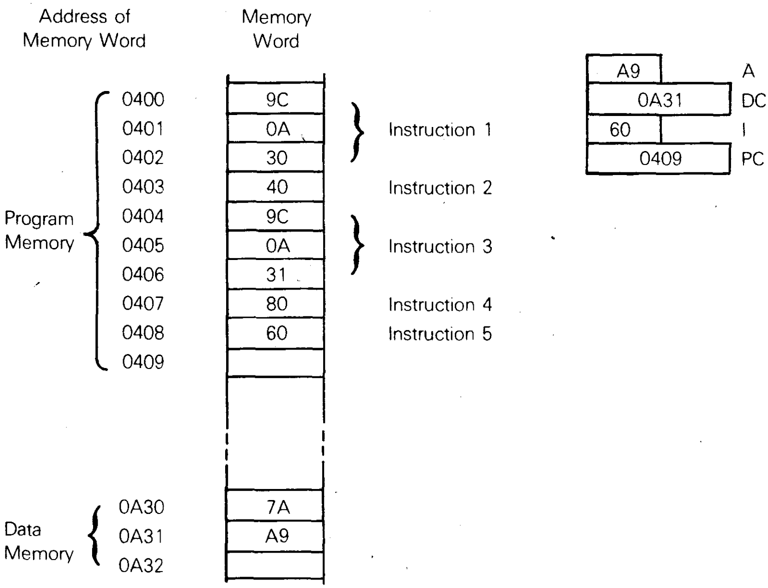
Instruction 4 has now completed execution.

If the sum in A were being returned to any memory word other than $0A31_{16}$, we would now have to execute another variation of Instruction 1 to load a data memory address into DC. But the Accumulator contents are to be stored in memory word $0A31_{16}$, and that is the memory word currently addressed by DC, so a "load data memory address" instruction is unnecessary. We continue directly to Instruction 5, which stores the contents of the Accumulator into data memory word $0A31_{16}$ via these two steps:

Step 1, fetch the instruction code in the usual way:



Step 2, store the Accumulator contents into the memory word addressed by DC:



Instruction 5 has completed execution, and the program is done.

THE ARITHMETIC AND LOGIC UNIT

Actual data manipulations within the CPU are handled by a group of discrete logic components referred to collectively as the Arithmetic and Logic Unit (ALU). An ALU must be capable of operating on binary data in memory word increments; in other words, an 8-bit microcomputer's ALU will operate on 8-binary digit data units. The ALU must have logic to perform the following operations:

1) Binary addition

2) Boolean operations

3) Complement a data word

4) Shift a data word one bit to the right, or to the left.

Any more complex data manipulation operation required of a Central Processing Unit may be built up from these few logic elements of the ALU.

THE CONTROL UNIT

It is the Control Unit (CU) that sequences the logic elements of the ALU in order to implement any required operation. The Control Unit, in turn, is driven by the contents of the Instruction register. In other words, the Instruction register contents are decoded by the Control Unit. In response to the bit pattern of the instruction code, the Control Unit generates a sequence of enable signals to flow data appropriately through the ALU and to enable ALU logic modules at the proper time. **The microcomputer CPU that results is illustrated in Figure 4-1.**

The "Buffer register" holds data that are transiently in the CPU. For example, when two data bytes are added (as in Instruction 4 of the binary addition example), the data word which is fetched from memory, to be added to the Accumulator contents, will be stored in the Buffer register.

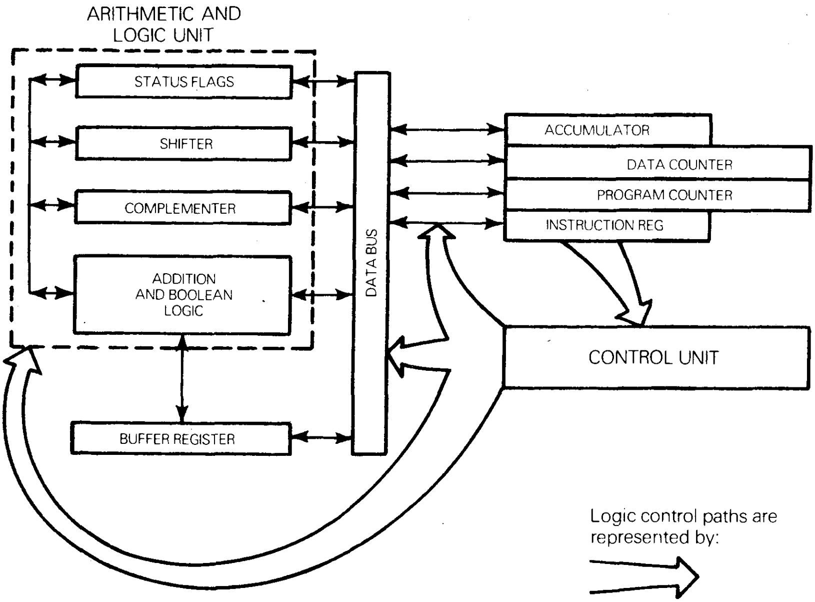


Figure 4-1. Functional Representation of a Control Unit

STATUS FLAGS

A CPU must have a set of single binary digit logic gates which are automatically set or reset to reflect the results of ALU operations. Each binary digit logic gate is called a status flag.

CARRY STATUS

We have already encountered two status flags in Chapter 3: the Carry and Intermediate Carry. In order to perform multibyte arithmetic, any carry out of the high order bit of two data words must be recorded in the Carry status, so that it may be propagated into higher order memory words:

```
 High order          Low order
    word                word

0 1 0 1 1 0 1 1     1 0 1 1 1 0 0 0
0 0 1 0 1 1 0 1     1 1 0 1 1 0 1 0 ── Carry status from
              1 ◄───────────────────   high order bit of
_______________     _______________    low order word
1 0 0 0 1 0 0 1     1 0 0 1 0 0 1 0

     High order bit position
```

The Carry status is also useful when performing multiword shift operations, as described in Chapter 6.

INTERMEDIATE CARRY STATUS

In order to perform BCD arithmetic, it is also necessary to record any carry out of the low order four bits of an 8-bit unit, since, as described in Chapter 2, each 4-bit unit of a byte encodes a separate decimal digit.

There are some additional statuses which may also prove useful when performing various types of data manipulation or decision making operations.

ZERO STATUS

A Zero status flag may be set to 1, to indicate that a data manipulation operation generated a zero result; this flag will be reset to 0 otherwise. A word of caution is required at this point. Most microcomputers and minicomputers have a Zero status flag. It is universally accepted that the Zero status flag will be set to 1 if a data manipulation operation generates a zero result, while the status flag is set to 0 for a non-zero result. In other words, the Zero status flag is universally set to the complement of the result condition.

WHEN STATUSES ARE MODIFIED

Another important point should be made concerning the Zero status flag, and most other status flags. **Those instructions which set or reset status flags, and those which do not, are carefully selected by microcomputer designers.**

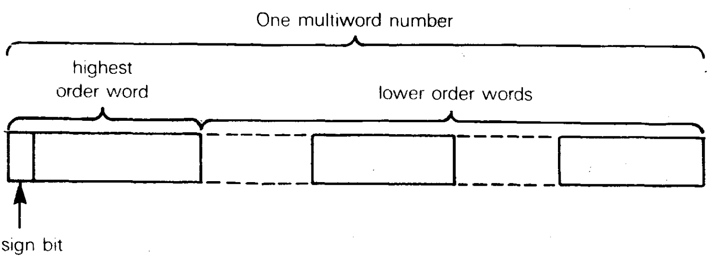
Consider the very obvious case of multibyte addition, as illustrated above. The low order words of two 2-word numbers are added, and the Carry status is set or reset to reflect any carry out of the high order bit of the low order words. The carry must be added to the low order bits of the two high order words of the two-word numbers. This means that the Carry status must be preserved while the two high order words are loaded into CPU registers. Clearly it would be disastrous to program logic if, when a word of data was loaded from memory, the Carry status was cleared to reflect the fact that the load operation did not generate a carry.

At this point, it is only important to remember that every instruction will not affect every, or for that matter any, status flags; moreover, the way in which status flags are set or reset is very important and is one of the most carefully thought out feature of any microcomputer CPU design. In other words, status flags do not necessarily represent conditions within the CPU now; they may well represent the results of selected key operations the last time these operations were performed.

SIGN STATUS

The use of the high order bit of a memory word as a sign bit, when performing signed binary operations, gives rise to two status flags. First, there is the Sign status, which is simply the contents of the sign bit (or its complement). The Sign status flag allows tests to be made for positive or negative numbers when memory words are being interpreted as signed binary data.

The sign bit is always the highest order bit of any number, single word or multiword:



To the microcomputer, however, the high order bit of every byte will be treated as a sign bit. Program logic must decide when to ignore the sign status and when to interpret it.

Then there is the overflow status. Recall that the microcomputer CPU is going to treat every binary addition alike — as a pure binary addition. If a carry is generated when adding two lower order words of two multiword numbers, then the carry is legal and simply reflects a carry into the next higher word of the sum. This is illustrated for the addition of two 4-word numbers, with eight bits per word:

OVERFLOW STATUS

```
                                  Magnitude of number
        Sign bits
                          ______________________________________
        First number      00100011   01011011   11001011   10111100
        Second number     00011101   10111001   11011000   11000111
            Valid carries:        1          1          1
                          ---------------------------------------
                   Sign:  01000001   00010101   10100100   10000011
```

The microcomputer CPU has no way of knowing whether a memory word is a single numeric entity or part of a multiword numeric entity and, if part of a multiword numeric entity, whether it is in the middle of the word or at one end of the word. This being the case, a carry generated as the result of an addition is, so far as the microcomputer logic is concerned, always perfectly valid.

When the high order bit of a data word is being interpreted as a sign bit, any carry out of the penultimate bit will represent an error overflow, that is, a result that will not fit in the allocated space. Consider a single, 8-bit data word, being interpreted as signed binary data:

```
                     Sign bit
                     High order data bit
   5A         0 1 0 1 1 0 1 0
 + D3        +0 1 1 0 1 0 1 1
 = -3B?      =1 1 0 0 0 1 0 1
```

A carry out of the high order data bit represents a result too big to fit into the allowed data space.

We must devise a strategy for identifying erroneous results of signed binary addition.

Does a carry out of the high order data bit always signal an error? Indeed no; consider (−2) + (−2) = (−4):

```
                     Sign bit
                     High order data bit
  (-2)        1 1 1 1 1 1 1 0
 +(-2)        1 1 1 1 1 1 1 0
 =(-4)        1 1 1 1 1 1 0 0
```

There is a carry out of the high order data bit

Although there is a carry out of the high order data bit, the result is −4, which is correct. We will use the symbol C_s to represent a carry out of the sign bit and C_p to represent a carry out of the high order data bit. What if C_s and C_p are both 0?

```
   2          0 0 0 0 0 0 1 0
  +2          0 0 0 0 0 0 1 0
  =4          0 0 0 0 0 1 0 0

       Cp=0      Cs=0
```

```
   11110100        (-0C)
 + 00001001      +   09
 = 11111101      =(-03)

 Cs = 0  <--  Cp = 0
```

So long as both C_s and C_p are zero, the answer is always correct.

Now consider some examples where C_s and C_p are both 1:

```
   1 0 0 0 1 0 1 1      (-75)
 + 0 1 1 1 1 0 0 1    +   79
   0 0 0 0 0 1 0 0    =(+ 4)

 1 <-- 1
 Cs    Cp
```

(Recall that 10001011 is -75_{16} because $+75_{16}$ is 01110101, the twos complement of which is 10001011.)

```
   1 1 0 1 1 0 0 0      (-28)
 + 0 1 0 1 1 0 0 1    +   59
   0 0 1 1 0 0 0 1    =(+31)

 1 <-- 1
 Cs    Cp
```

```
   1 1 0 0 0 1 1 1      (-39)
 + 1 1 1 0 0 1 1 0    +(-1A)
   1 0 1 0 1 1 0 1    =(-53)

 1 <-- 1
 Cs    Cp
```

When C_s and C_p are both 1, the answer is always correct.

When C_s and C_p differ, that is, either one or the other, but never both are 1, the answer is always in error:

```
   0 1 0 0 0 1 0 1        45
   0 1 1 0 0 1 1 1    +   67
   1 0 1 0 1 1 0 0    =(-54) ?

 0 <-- 1
 Cs    Cp
```

```
   1 0 0 1 0 0 1 0      (-6E)
   1 0 1 0 0 1 0 0    +(-5C)
   0 0 1 1 0 1 1 0    =  +36  ?

 1 <-- 0
 Cs    Cp
```

Our strategy for setting and resetting the Overflow status is therefore clear. When carries out of the sign and penultimate bits are the same (C_p and C_s are both 0 or both 1), the Overflow status will be set to zero. When these two carries differ, the Overflow status will be set to 1, indicating that the answer overflowed the answer space and is therefore wrong.

OVERFLOW STATUS SET STRATEGY

Stated another way, **the Overflow will be the Exclusive OR of the carries out of the sign and penultimate bits:**

$$\text{OVERFLOW} = C_s \oplus C_p$$

The Parity status is the only other status which is worth mentioning at this time. This flag, if present, is set to 1 each time a data transfer operation detects a data byte with the wrong parity. Clearly this status will be ignored most of the time, since it is only meaningful when the contents of a memory word is being interpreted as a character code.

PARITY STATUS

INSTRUCTION EXECUTION

We have described how a microcomputer CPU may interpret the contents of a memory word as an instruction code. But this leaves a number of unanswered questions. What maximum or minimum number of logical events constitutes an instruction? What occurs within the CPU during the course of an instruction's execution? And what external support logic does the CPU demand?

In answering these questions, we introduce a very critical microcomputer concept, and one of the key differences between the minicomputer and the microcomputer.

In the world of minicomputers, the most important feature to look for in an instruction set is the versatility of operations performed by the CPU in response to each instruction code. This is reasonable for minicomputers; minicomputers are frequently called upon to perform varied and varying tasks, and programming may be an ongoing, major expense.

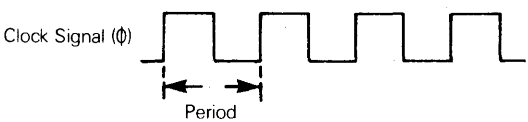
In the world of microcomputers, this question is far more important: what does the CPU demand of external logic?

Complex instructions usually demand complex logic external to the CPU. This is of no concern to the minicomputer user, who buys CPU plus all external logic, packaged in a single box. This is of great concern to the microcomputer user, who must interface his logic, often directly to the CPU. If a microcomputer costs somewhere between $5 and $100, the entire economics of using the microcomputer will evaporate unless the interface logic demanded by the microcomputer CPU is also inexpensive.

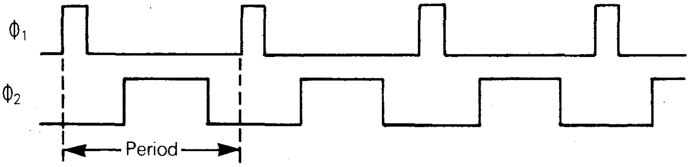
Let us therefore examine how an instruction is executed, and then return, at the end of this chapter, to the differences between minicomputer and microcomputer.

INSTRUCTION TIMING

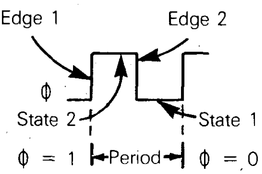
As with all digital logic, operations within a microcomputer CPU are controlled by a crystal clock, with a period which may vary from as little as 100 nanoseconds to as much as a microsecond. We will refer to this clock signal using the symbol ϕ:



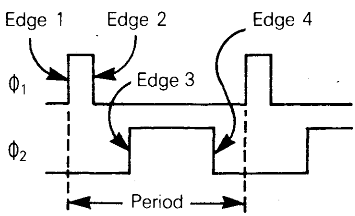
While the crystal must be external to the CPU chip, the logic which generates the clock signal may or may not be on the same chip as the CPU. Moreover, depending on how the CPU has been designed, the timing signal may be a straightforward single signal, as illustrated above, or it may consist of a more complex interaction of signals. Here is one possible combination of two signals, identified by the symbols ϕ_1 and ϕ_2:



The simple signal provides two edges and two states per period:



The more complex signals provide four edges and three states per period:



State 1: $\phi_1 = 0$ $\phi_2 = 0$
State 2: $\phi_1 = 0$ $\phi_2 = 1$
State 3: $\phi_1 = 1$ $\phi_2 = 0$

In this chapter we will use the simple signal ϕ.

INSTRUCTION CYCLES

The execution of every instruction, by any microcomputer, may be divided into two parts: the instruction fetch and the instruction execute. This was illustrated earlier in this chapter for the six-step binary addition example. Recall that every instruction starts with the instruction code being loaded into the Instruction register. We will refer to this operation as an instruction fetch.

INSTRUCTION FETCH

During the instruction fetch, CPU logic outputs the contents of the Program Counter register, along with appropriate control signals specifying that external logic is to return the contents of the memory word addressed by the Program Counter. So far as external logic is concerned, this is simply a read operation.

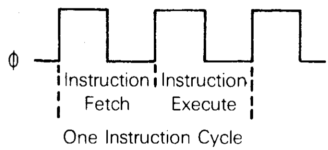
The contents of the memory word, when received by the CPU, is stored in the Instruction register, and thus gets interpreted as an instruction code.

While external logic is responding to the instruction fetch, the CPU uses its own internal logic to add 1 to the contents of the Program Counter. The Program Counter now points to the memory word following the one from which the current instruction code was fetched.

INSTRUCTION EXECUTE

Once the instruction code is in the Instruction register, it triggers a sequence of events controlled by the Control Unit; this sequence of events constitutes the instruction execution.

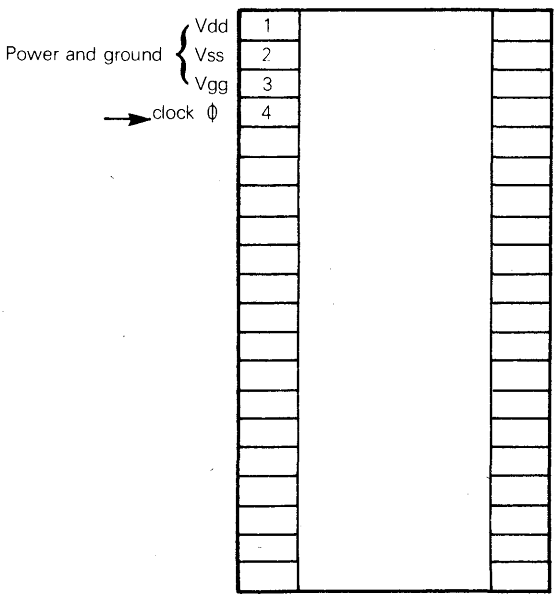
Two clock periods will be used to execute an instruction; one will time the instruction fetch, the next will time the instruction execute:



CPU PINS AND SIGNALS

Next consider the signals via which the CPU will communicate with external logic. The 40-pin DIP, being the most popular among today's microcomputers, is the one we will adopt — which means that 40 signals may be input and/or output, including the clock, power and ground.

The way in which the 40 pins of the DIP are used constitutes one of the most variable features of microcomputers, but they all begin along these lines:



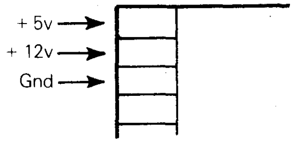
Vdd is the current drain connection, or power input.

Vss is the current source, or ground.

Vgg is the gate voltage; it is not required in all LSI devices.

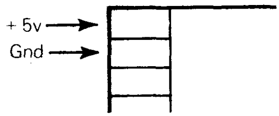
POWER SUPPLIES

Many devices will show power connections much more simply, in one of the following ways:



GROUND

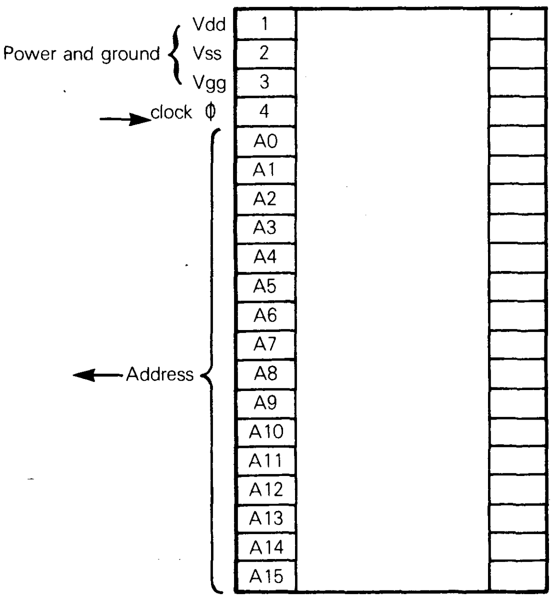
In this case the device requires two power supplies, + 5v and + 12v, plus a single ground. Frequently a single power supply will suffice:



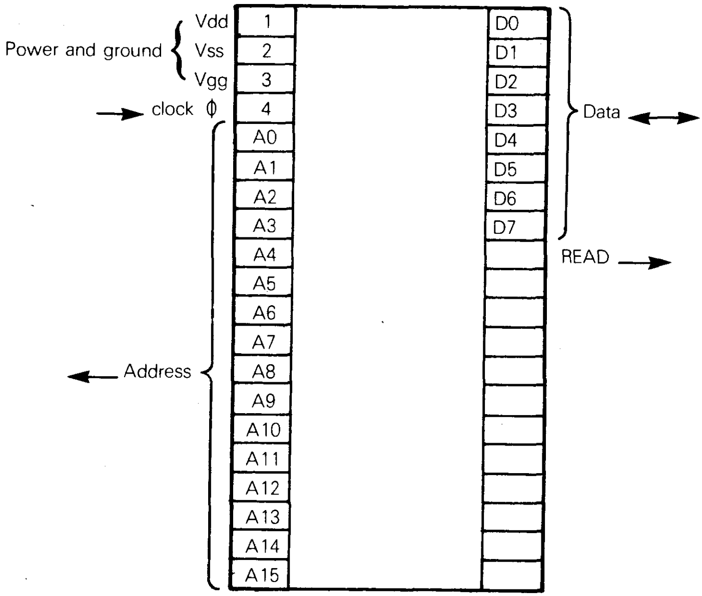
INSTRUCTION FETCH SIGNALS AND TIMING

The first step, when executing any instruction, is the instruction fetch; that is, in reality, a memory read. It requires a memory address to be output, and a data word to be input in response.

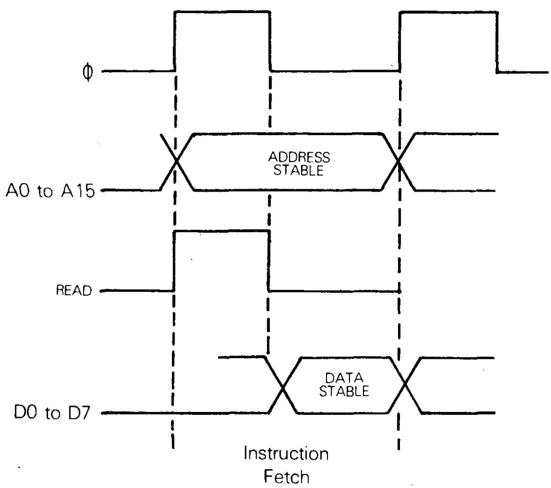
If the memory address can range from 0 to 65,535, sixteen address pins will be required, one for each binary digit of the address:



All data will enter and leave the microcomputer CPU via eight bi-directional signals. A READ control signal, indicating that data must be input to the CPU, completes the requirements for the instruction fetch:



The following timing diagram defines the instruction fetch sequence, as controlled by the CPU:



We will now turn our attention to the requirements for instruction execution.

Consider the six-step binary addition program which was described at the end of Chapter 3. There are four separate and distinct types of instructions within the program. They are:

1) **Load a memory address into the Data Counter (Instructions 1 and 3).**
2) **Fetch the contents of the data word addressed by the Data Counter and store it in the Accumulator (Instruction 2).**
3) **Fetch the contents of the data word addressed by the Data Counter, add it to the contents of the Accumulator, and store the result in the Accumulator (Instruction 4).**
4) **Store the contents of the Accumulator in the memory word addressed by the Data Counter (Instruction 5).**

Let us examine how each of these four instruction types are executed, in terms of instruction cycles and control signals output by and input to the CPU.

MEMORY READ SIGNALS AND TIMING

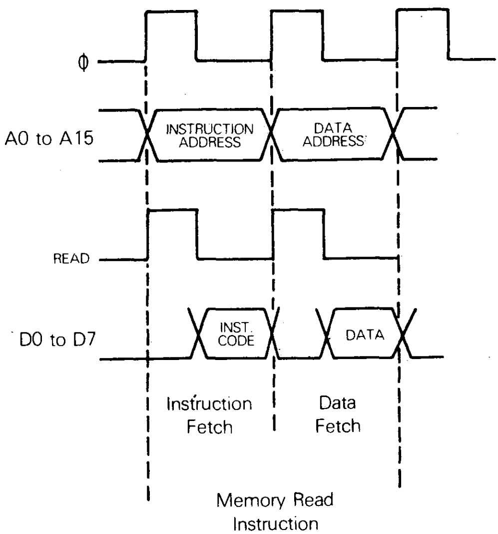
Instruction 2 is the simplest, so we will begin with it.

Like all instructions, it begins with an instruction fetch.

The Control Unit decodes the instruction code 40_{16}, and in response causes a data word to be fetched from memory. In reality, since the data is in a memory word on a memory device, all the CPU can really do is generate signals at its pins; logic external to the CPU must respond to these signals if the data fetch is to be accomplished.

As seen by external logic, signals generated by the CPU to fetch data are identical to signals generated for the instruction fetch.

Thus **timing for a Memory Read instruction is as follows:**



These are the only differences between the instruction fetch and the data fetch cycles:

1) During the instruction fetch cycle, the address output on A0 - A15 is the contents of the PC register; during the data fetch cycle, it is the contents of the Data Counter.

2) During the instruction fetch, data input is stored in the Instruction register; during the data fetch, it is stored in the Accumulator.

EXTERNAL LOGIC REQUIREMENTS

This simple scheme demands very little of the external logic. If READ is high when ϕ is high, then memory circuits must decode A0 - A15. The selected memory module must extract the contents of the addressed memory word and make sure it is at the microcomputer data pins when ϕ is low.

What the microcomputer CPU demands of external logic during a read operation is standard, simple logic that is part of any memory device; but we will defer this discussion to Chapter 5, and continue with CPU signals and timing for the ADD instruction.

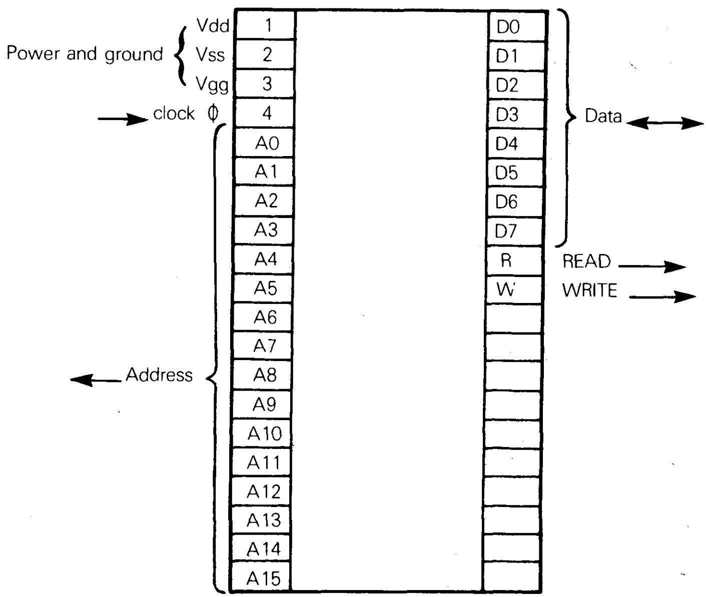
ADD OPERATION SIGNALS AND TIMING

In order to perform an add, the CPU fetches the contents of a memory word, exactly as it did for a Memory Read instruction; however, for the Add instruction, the fetched data are added to the contents of the Accumulator; by contrast, data fetched during a Memory Read instruction are deposited in the Accumulator unchanged.

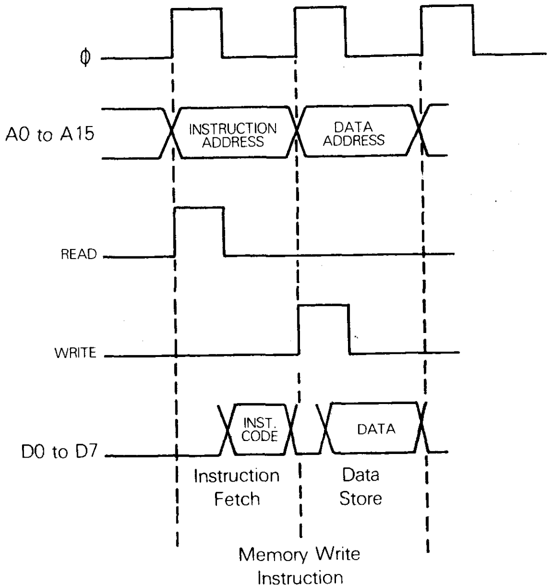
As seen by external logic, therefore, there is no difference between the signals generated for an Add or a Memory Read instruction.

MEMORY WRITE SIGNALS AND TIMING

Instruction 4 causes the contents of the Accumulator to be stored in the memory word addressed by the Data Counter; this is called a Memory Write instruction. **As seen by external logic, the only difference between the signal sequences for a Memory Read and a Memory Write is that a WRITE signal must go high, instead of a READ signal, when ϕ is high. We must therefore add a WRITE signal to our CPU device:**



Timing for a Memory Write instruction is as follows:



LOAD DATA COUNTER SIGNALS AND TIMING

Instruction 1 loads a memory address into the Data Counter. This instruction occupies three memory words, one for the instruction code and two more for the memory address.

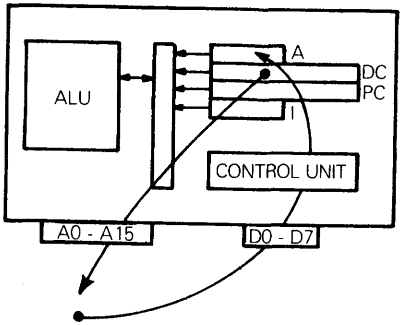
Notice that the Load Data Counter instruction is equivalent to two memory reads, with these differences:

1) Both Load Data Counter memory reads are specified by one $9C_{16}$ instruction code. By contrast, the 40_{16} Memory Read instruction code triggers just one memory read.

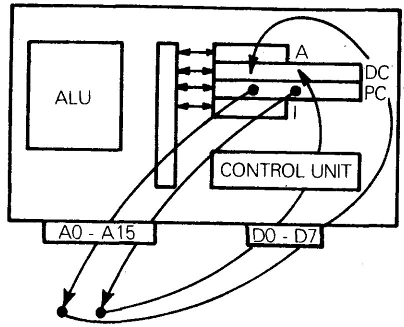
2) The Load Data Counter instruction fetches data from memory words whose addresses come from the Program Counter. For a Memory Read instruction, the Data Counter provides the data memory address.

3) Data read from memory by the Load Data Counter instruction is stored in the upper and lower halves of the Data Counter. For a Memory Read instruction, the fetched data is stored in the Accumulator.

The differences between the Load Data Counter and the Memory Read instructions may be contrasted as follows:

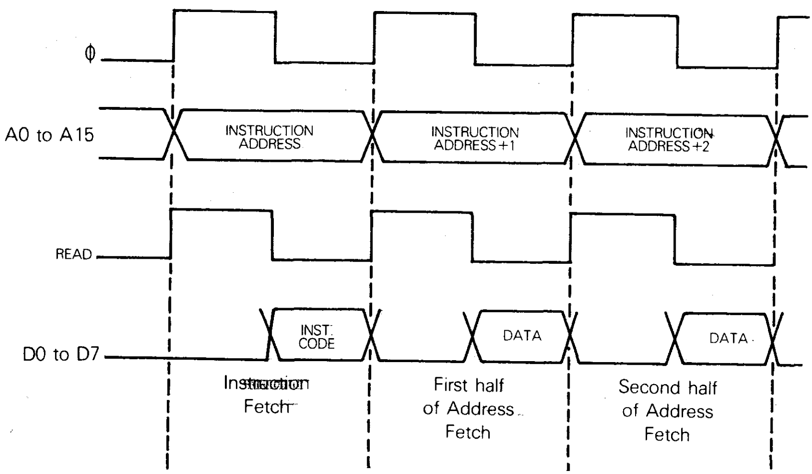


MEMORY READ

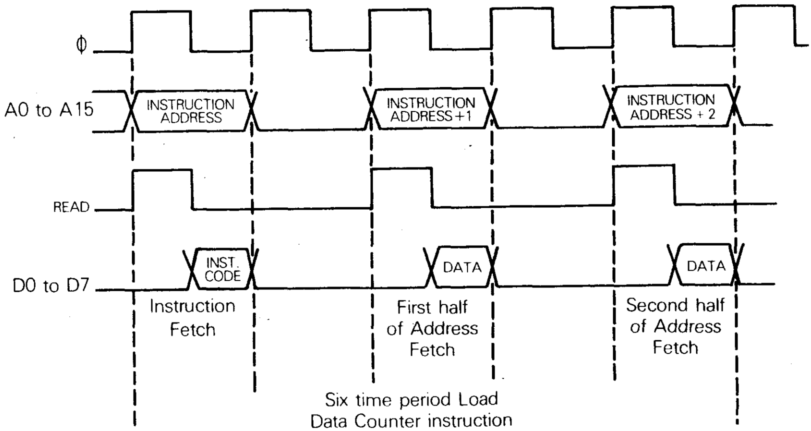


LOAD DATA COUNTER

While the logic operations internal to the CPU are completely different for the Load Data Counter and the Memory Read instructions, the external timing and signal sequences are remarkably similar. A very smart microcomputer CPU could execute the Load Data Counter instruction in three time periods, as follows:



Some microcomputers do not try to be so clever. In order to simplify CPU logic, the Program Counter contents are only output, as an address, during the first clock period of an instruction. The Data Counter contents, likewise, are only output during the second clock period of an instruction. This simpler CPU will require six clock periods to execute the Load Data Counter instruction:



HOW MUCH SHOULD AN INSTRUCTION DO?

Let us now consider ways in which the instructions described in this chapter may be made simpler or more complex.

The Load Data Counter instruction loads the two 8-bit words which follow the instruction code into the Data Counter. This instruction could be broken up into two instructions, one to load the low order byte of the Data Counter, the other to load the high order byte of the Data Counter. Let us compare the two forms of the Load Data Counter instruction:

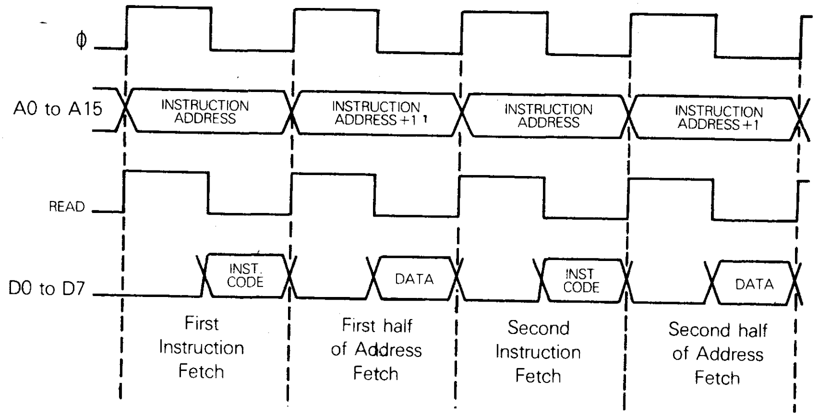
9C	Load $0A30_{16}$ into the Data Counter
0A	
30	

One three-byte Load Data Counter instruction

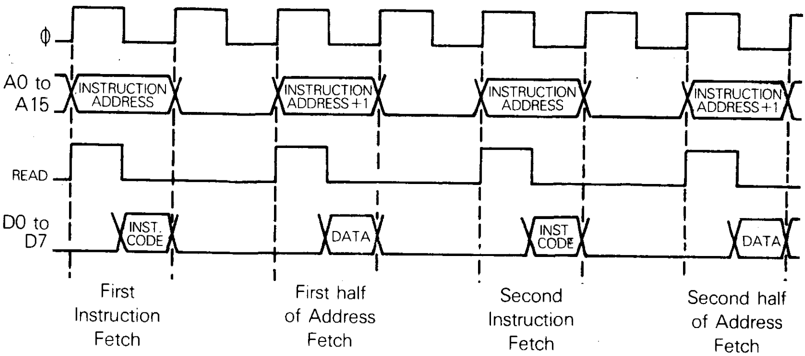
9D	Load $0A_{16}$ into Data Counter High order byte
0A	
9E	Load 30_{16} into Data Counter Low order byte
30	

Two two-byte Load Data Counter Instructions

The three-byte Load Data Counter instruction's possible signals and timing have been illustrated. The two 2-byte instructions could each execute in two time periods or in four time periods. Executing in two time periods, signals and timing for each instruction would be as follows:

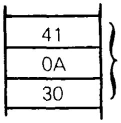


Timing and signals, when executing in four time periods, would be as follows:



Breaking the three-byte Load Data Counter instruction into two 2-byte instructions does not simplify the demands placed on external logic by the CPU; but it does make the microcomputer's Control Unit simpler, as we will demonstrate later in this chapter when describing microprogramming.

Now consider combining the Load Data Counter and Memory Read instructions as follows:

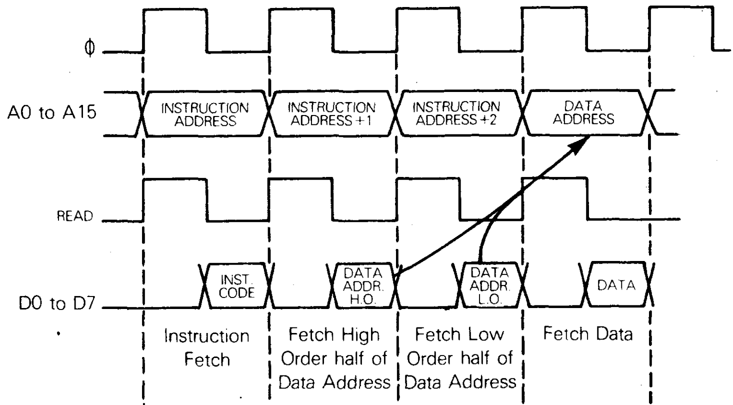


Load into the Accumulator the contents of the memory word addressed by the next two words of this three-word instruction.

As illustrated above, 41_{16} is the instruction code specifying this three-byte Memory Read instruction; $0A30_{16}$ is the address of the memory word whose contents is to be read into the Accumulator. Instructions that specify the memory address to be referenced, as this three-byte Memory Read instruction does, are said to have direct memory addressing.

DIRECT ADDRESSING

Signals and timing for the three-byte Memory Read instructions could take one of many forms. Here is the most compact possibility:



Three-byte Memory Read Instruction Execution

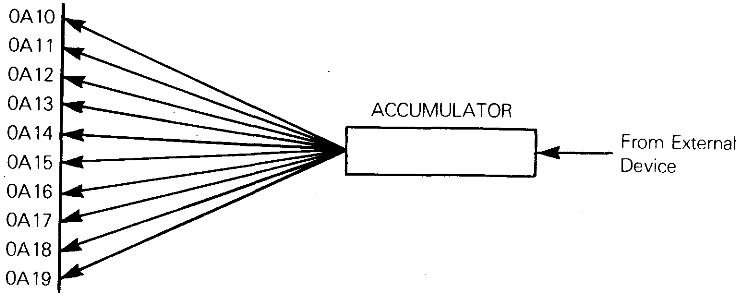
To a minicomputer designer, combining the Load Data Counter instruction with the Memory Read — or with any memory reference instruction — is obvious.

To the microcomputer designer, the automatic virtues of direct addressing are not so obvious, for the immediately apparent reason that direct addressing instructions require more complex Control Unit logic; this will be illustrated later in this chapter when microprogramming is described.

But there is less immediately apparent reason why direct addressing is not obviously desirable; whereas most minicomputer programs are stored and executed out of RAM, most microcomputer programs are stored and executed out of ROM; that means direct addressing can only be used when the data address will not change.

REFERENCING DATA TABLES

Consider an elementary instruction sequence which receives data input from an external device, then stores the data into a number of consecutive memory words:



RAM data memory words in the above illustration constitute a "data table." The beginning address, $0A10_{16}$, has been arbitrarily selected; any other address would do as well.

Ignoring, for the moment, the question of how many data words are to be stored in the data table, the following instruction sequence would fill the data table:

Arbitrarily selected addresses	Program Memory	
0280	9C	Load the address $0A10_{16}$ into the Data Counter
0281	0A	
0282	10	
0283	08	Input a byte from an external device to Accumulator
0284	60	Store Accumulator contents in memory
0285	E3	Increment Data Counter
0286	BC	Reset Program Counter to 0283
0287	83	

A straightforward Load Data Counter instruction, stored in program memory words 0280_{16}, 0281_{16} and 0282_{16}, loads the address $0A10_{16}$ into the Data Counter; this is the address of the first word in the data table.

The instruction code 08_{16}, in program memory word 0283_{16}, causes a byte of data to be input to the Accumulator from an external device.

The instruction code 60_{16}, in program memory word 0284_{16}, is a simple Store Memory instruction; it causes the contents of the Accumulator to be stored in the data memory word addressed by the Data Counter; initially that is the first word of the data table, with address $0A10_{16}$.

PROGRAM LOOP

The next two instructions, located in program memory words 0285_{16}, 0286_{16} and 0287_{16}, increment the Data Counter contents (to address the next word of the data table), **then change the value in the Program Counter to 0283_{16};** execution now returns to the instruction which brings the next data word into the Accumulator from the external device. We have established a program loop — a group of instructions that continuously get re-executed; a slightly different task is performed on each re-execution, because the Data Counter contents is incremented on each pass.

Four instructions, occupying five memory bytes, can fill a data table, whatever the length of the data table may be!

Using direct addressing, this program loop could not be executed. We would have:

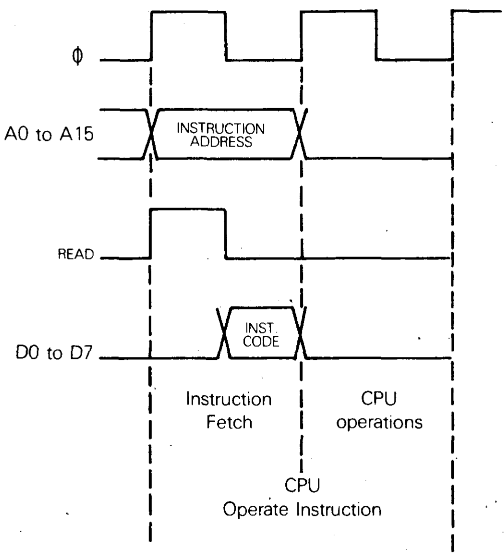
Address	Program Memory	
0280	08	Input a byte from an external device to Accumulator
0281	61	Store Accumulator contents in data memory word addressed by the second and third instruction bytes
0282	0A	
0283	10	
0284		Increment what?
0285		
0286		

The data table is addressed by the second and third bytes of the memory-store-with-direct-addressing instruction. This address cannot be incremented if it is going to reside in ROM! Minicomputers have a solution to this problem, of course (we shall see what the solution is in Chapter 6), but the solution adds complexity to microcomputers and the complexity may bring with it more cost than savings.

Two of the new instructions in the program loop need to be described further at this point.

CPU OPERATE INSTRUCTIONS

The Increment Data Counter instruction simply causes the contents of the Data Counter to be increased by 1; following the instruction fetch, logic external to the CPU is idle:



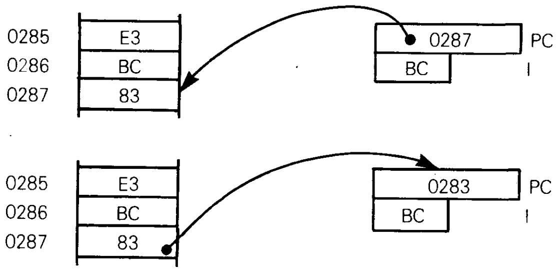
BRANCH INSTRUCTION

JUMP INSTRUCTION

The instruction in program memory words 0286_{16} and 0287_{16} actually changes the contents of the Program Counter, and thus changes the sequence in which instructions are executed. This is referred to as a Branch or Jump instruction.

Branch instructions have many variations. A two-word version is illustrated in the program loop; the contents of the second instruction word is loaded into the low order half of the Program Counter as follows:

ABSOLUTE BRANCH



BRANCHING AT PAGE BOUNDARY

The problem with this variation of the Branch instruction is that it will not work if the Program Counter high order byte gets incremented. For example, suppose the program loop was stored in memory as follows:

	Program Memory	
02FC	9C	Load Data Counter
02FD	0A	
02FE	10	
02FF	08	Input from External Device
0300	60	Memory Write
0301	E3	Increment Data Counter
0302	BC	Branch to FF
0303	FF	

Branch to FF_{16} would branch to $03FF_{16}$, not to $02FF_{16}$, because the high order byte of the Program Counter got incremented between the input and memory write instructions.

There are two ways around this problem.

First, we can have a three-word Branch instruction which changes both halves of the Program Counter.

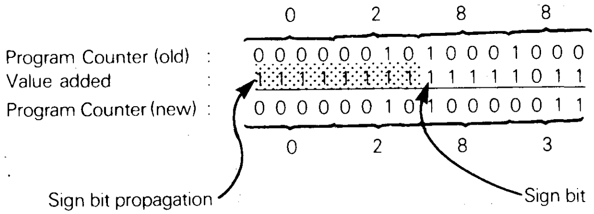
PROGRAM RELATIVE BRANCH

Second, we can add the contents of the second Branch instruction word to the Program Counter, designing the CPU so that it interprets the second Branch instruction word as a signed binary number. Referring to the program loop, after the Branch instruction had executed, the Program Counter would normally contain 0288_{16}; to change this value to 0283_{16}, 5 must be subtracted. The twos complement of 5 is:

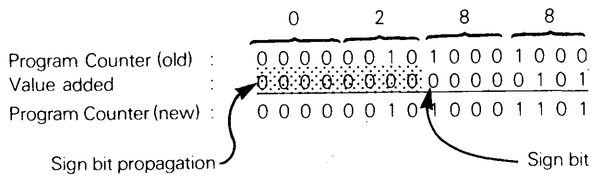
1 1 1 1 1 0 1 1

or FB_{16}. This is the value that would be stored in program memory word 0287_{16}. **Adding an 8-bit value from memory to the 16-bit contents of the Program Counter using signed binary arithmetic is not a problem; CPU logic simply propagates the sign bit through the high order half of the value to be added to the Program Counter.** In this case we have:

SIGN PROPAGATION



Adding 5 to the Program Counter contents would proceed as follows:



This is referred to as a Program Relative branch.

MICROPROGRAMMING AND THE CONTROL UNIT

Let us now examine how the Control Unit decodes instruction codes.

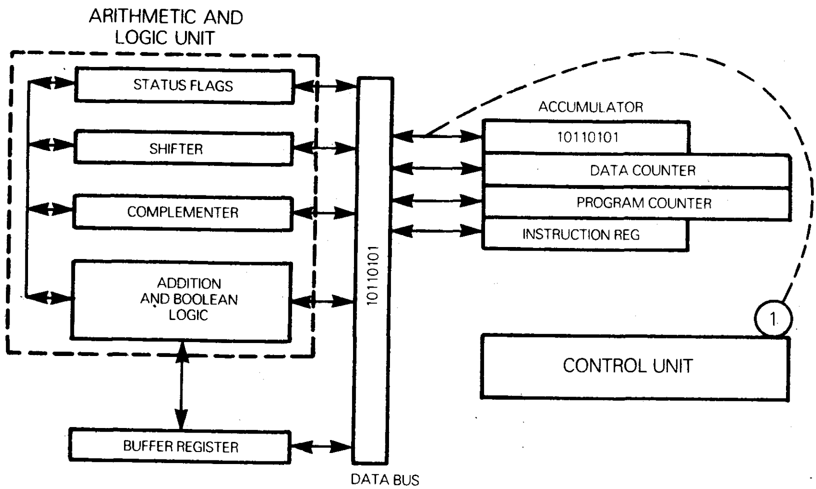
A microcomputer CPU may be illustrated functionally, as in Figure 4-1, but in reality, the CPU consists of a number of logic elements, activated by sequences of enable signals.

The Complementer, for example, is latently able, at any time, to complement the contents of eight data latches within the logic of the complementer circuits. A single enable signal, emanating from the Control Unit, will activate this logic sequence.

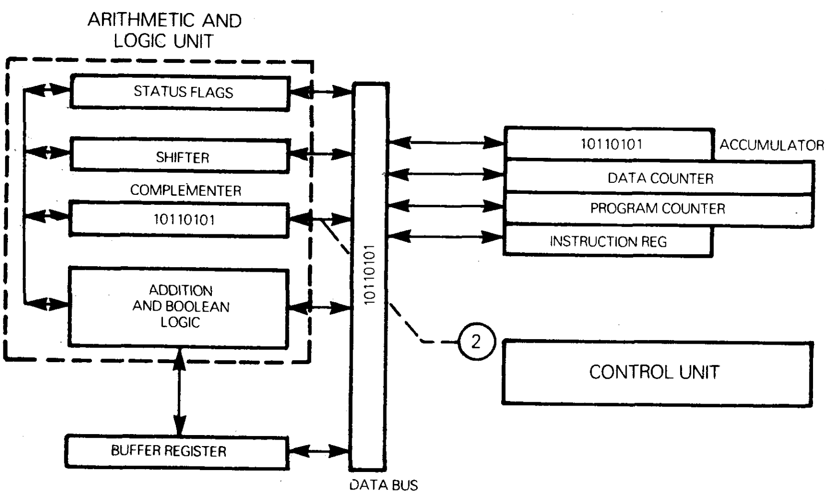
However, complementing eight data latches within the Complementer serves no useful purpose. We want to complement the contents of the Accumulator, and that means moving the contents of the Accumulator to the Complementer, then, after enabling complementer logic, returning the results to the Accumulator.

Complementing the contents of the Accumulator therefore requires these five steps:

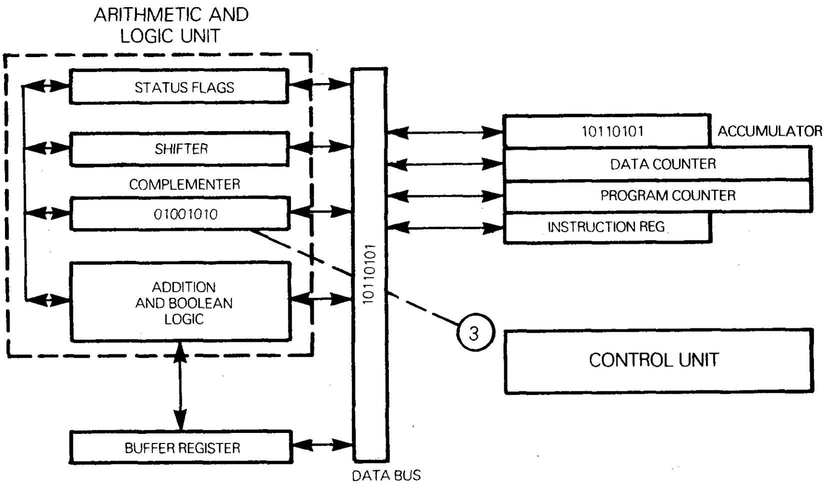
1) Move the contents of the Accumulator to the Data Bus:



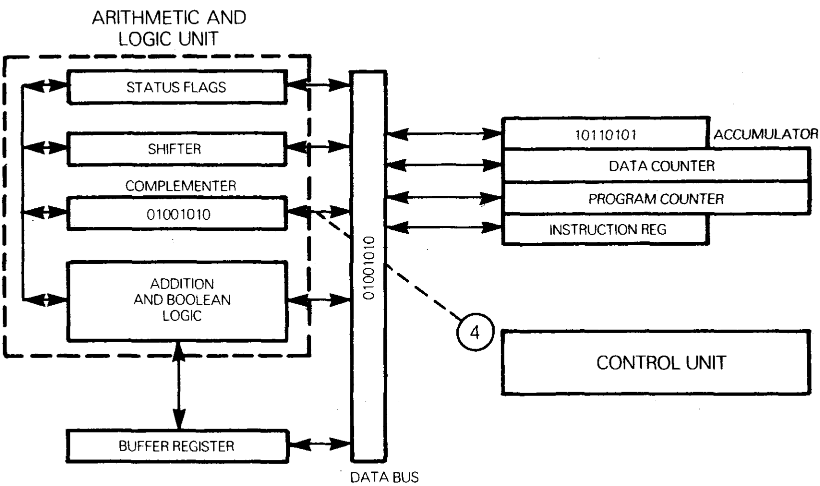
2) Move the contents of the Data Bus to the Complementer:



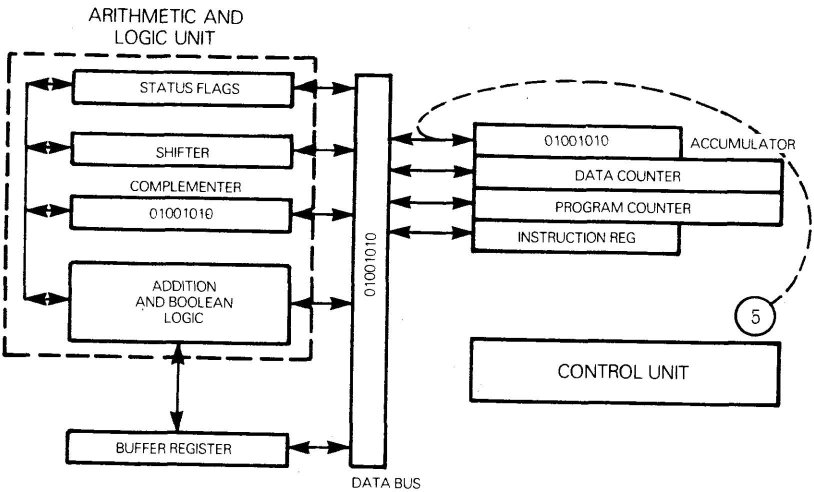
3) Activate Complementer logic:



4) Move the contents of the Complementer to the Data Bus:



5) Move the contents of the Data Bus to the Accumulator:



Each of these five steps is referred to as a microinstruction. Each microinstruction is enabled by a signal from the Control Unit. By outputting the appropriate sequence of control signals, **the Control Unit can sequence any number of microinstructions, to create a macroinstruction,** which is the accepted response of the CPU to an assembly language instruction code.

MICRO-INSTRUCTIONS

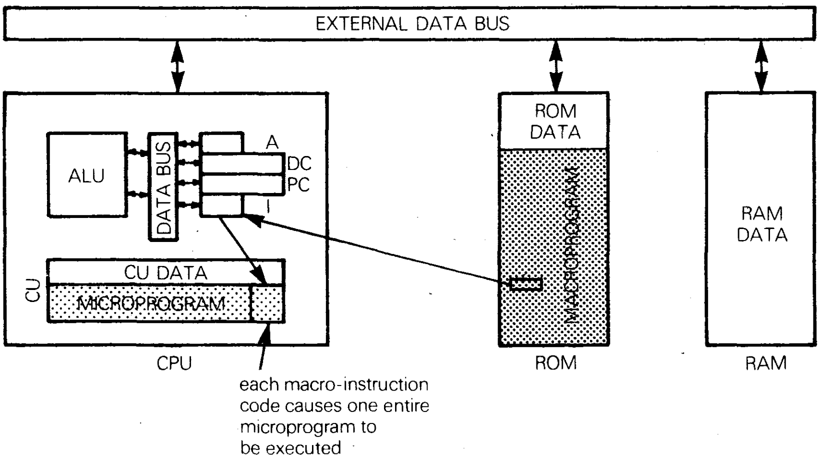
MACRO-INSTRUCTIONS

In order to complement the Accumulator contents, the Control Unit must contain five binary codes, each of which triggers an appropriate control signal (or signals). This sequence of binary codes within the Control Unit is referred to as a microprogram. Generating the sequence of binary codes that are stored within the Control Unit is referred to as microprogramming.

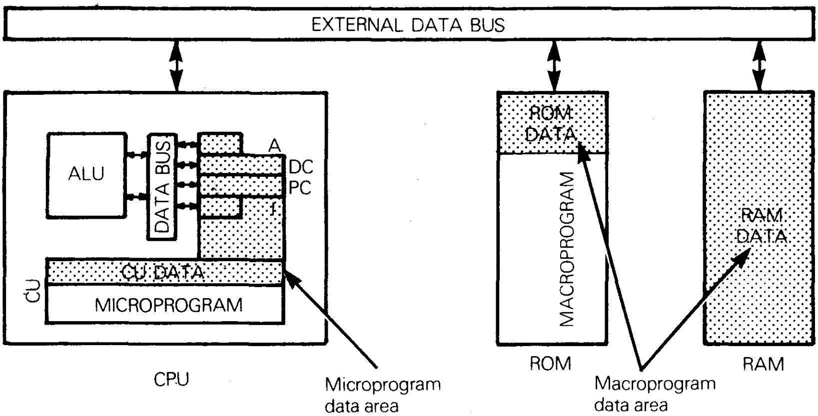
MICRO-PROGRAMS

There is a close parallel between microprogramming and assembly language programming.

A microprogram is stored as a sequence of binary digits in the Control Unit. An assembly language program is stored as a sequence of binary digits, usually in a ROM memory. The assembly language program is referred to as a macroprogram. Each instruction code of the macroprogram initiates execution of an entire microprogram, as stored in the Control Unit:



A microprogram stored in the Control Unit has a data memory, which consists of the CPU registers, plus a data storage internal to the Control Unit. A macroprogram has a data area, which consists of ROM memory for constant data, plus RAM memory for variable data:



Individual instructions of a microprogram implement a small logic sequence within the logic of the CPU. Individual instructions of a macroprogram cause an entire microprogram to be executed, thus implementing a whole sequence of operations within the CPU.

MACRO-INSTRUCTION COMPLEXITY

The complexity of operations associated with any macroinstruction is a direct function of the size of the microprogram whose execution the macroinstruction initiates. There are no logical breakpoints or levels at which a microprogrammer must terminate the microprogram which will be executed in response to any macroinstruction code. Of course, complex microprograms require large Control Units. A simple microcomputer may have a small Control Unit and therefore may be forced to execute very simple macroinstructions. Some large computers have no assembly language, but in response to a single macroinstruction code, execute complex sequences of events involving logic throughout the computer system.

MICRO-PROGRAMMABLE MICROCOMPUTER

The Control Unit of every microcomputer is in reality nothing but a microprogram. If you, the user, are able to create or modify the microprogram within the Control Unit, then the microcomputer is said to be "microprogrammable". If the Control Unit microprogram is designed by the microcomputer logic designer, and then becomes an unalterable part of the CPU chip, the microcomputer is not microprogrammable.

In this book we are going to describe these two separate and distinct classes of microcomputer product:

1) **The microprocessor based microcomputer which gives you access to a Central Processing Unit, but not to the Control Unit.** You sequence CPU logic using macroinstructions, referred to collectively as an Assembly Language instruction set. You cannot microprogram this class of microcomputer product; nonetheless, a basic understanding of microprogramming will help you understand the tradeoffs that every microcomputer designer must evaluate when putting together an instruction set.

MACROLOGIC

MICRO-PROCESSOR SLICE

2) **The "microprocessor slice" or "macrologic" based microcomputer.** Here you are presented with CPU "building blocks", which you must tie together with a microprogram.

MICROPROCESSOR BASED MICROCOMPUTERS

First we will describe microprocessor based microcomputers.

Let us identify an arbitrary set of control signals, as illustrated in Figure 4-2. Our microcomputer Control Unit will generate these control signals to implement macroinstructions. Tables 4-1, 4-2 and 4-3 describe these control signals.

When compared to the ingenuity of real chip slice microinstruction codes, Tables 4-1, 4-2 and 4-3 represent a somewhat simplistic and inflexible CPU logic organization. Nevertheless, they will make real chip slice architecture easier to understand by clarifying the goals that the chip slice logic designer must attain.

The control signals described in Tables 4-1, 4-2 and 4-3 do not allow the Control Unit to perform all of the operations which will be needed to support assembly language instructions. For example, nothing has been said about how the READ and WRITE control signals will be generated, or how the four status latches C (Carry), O (Overflow), S (Sign) and Z (Zero) will be handled. One primitive scheme for handling these problems is to trap microinstructions that attempt to set both C0 and C1 to 1; this is an impossible condition, since it specifies data moving on and off busses simultaneously. **If C0 and C1 are both 1, they will be output as 0, and the remaining signals, C2 through C8, will be interpreted as specifying the following five different classes of internal Control Unit operations:**

1) If C2 through C8 are all 0, then the status latches Z, S, O and C, in the ALU, will have their condition recorded in the CU DATA buffer.

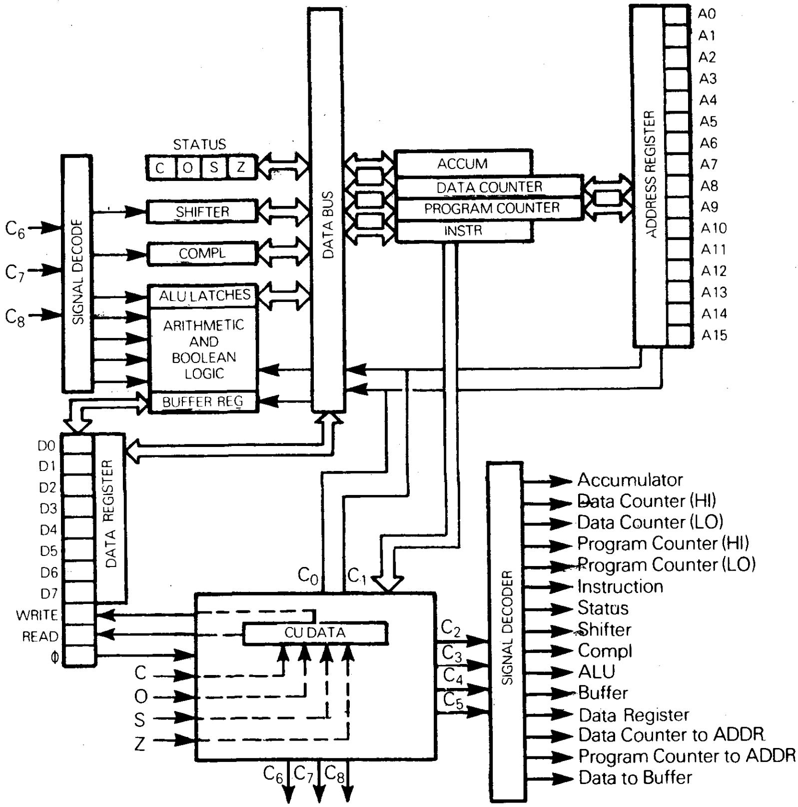


Figure 4-2. Control Unit Signals For a Simple Microcomputer

Table 4-1. Control Unit Signals

SIGNAL	FUNCTION
C0,C1	C0=0, C1=0; No data moved onto, or off Data Bus or Address Register C0=1, C1=0; Data moved onto Data Bus, or Address Register C0=0, C1=1; Data moved off Data Bus, or Address Register C0=1, C1=1; Microinstruction will be trapped within the Control Unit (see Table 4-4)
C2,C3, C4,C5	When C0=1 C1=0, or C0=0 C1=1, these four signals are decoded to select specific data flow, as specified in Table 4-2.
C6,C7,C8	These three signals are decoded to select ALU operations as specified in Table 4-3.
WRITE, READ	Direct connections from the input pins to two CU data bits.
Φ	Clock signal input to CU
C,O,S,Z	Four status bits, directly connected to four CU data bits.

Table 4-2 Data Flow Select When C0=1 Or C1=1

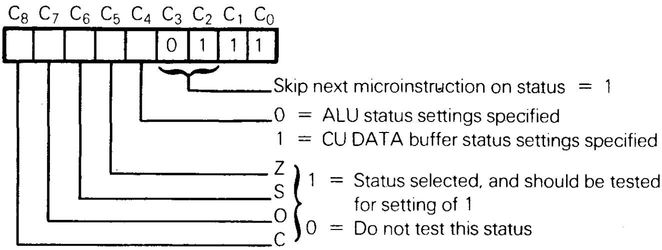
C_5	C_4	C_3	C_2	FUNCTION
0	0	0	0	Accumulator ⟷ Data Bus select
1	0	0	0	Data Counter high order byte ⟷ Data Bus select
0	1	0	0	Data Counter low order byte ⟷ Data bus select
1	1	0	0	Program Counter high order byte ⟷ Data Bus select
0	0	1	0	Program Counter low order byte ⟷ Data Bus select
1	0	1	0	Instruction Register ⟷ Data Bus select
0	1	1	0	Status Register ⟷ Data Bus select
1	1	1	0	Shifter ⟷ Data Bus select
0	0	0	1	Complementer ⟷ Data Bus select
1	0	0	1	ALU latches ⟷ Data Bus select
0	1	0	1	ALU Buffer ⟷ Data Bus select
1	1	0	1	Data Register ⟷ Data Bus select
0	0	1	1	Data Counter ⟷ Address Register select
1	0	1	1	Program Counter ⟷ Address Register select
0	1	1	1	Data Register ⟷ Buffer Register
1	1	1	1	Not Used

Table 4-3. ALU Select Signals

C_8	C_7	C_6	FUNCTION
0	0	0	Select shifter logic
1	0	0	Select complementer logic
0	1	0	Select addition logic*
1	1	0	Select AND logic*
0	0	1	Select OR logic*
1	0	1	Select XOR logic*
0	1	1	Increment ALU latches
1	1	1	No ALU operation

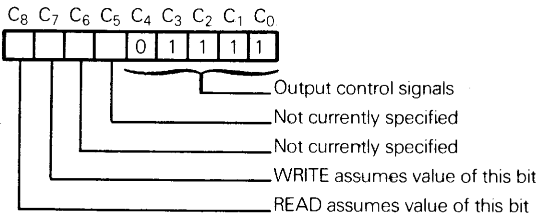
*Operation is performed on contents of ALU latches and Buffer register. Result appears in ALU latches.

2) If C2 and C3 are 1 and 0, then C5, C6, C7 and C8 will be interpreted as corresponding to the Z, S, O and C statuses, respectively. If C4 is 0, the status conditions in the ALU are referenced; if C4 is 1, the status conditions stored in the DATA buffer are referenced. C5, C6, C7 and C8 will each be checked for a 1 value. If a 1 is found, then the corresponding status will be checked. If the corresponding status has a value of 1, then the next microinstruction will be skipped. This use of the nine controls C_0 through C_8 may be illustrated as follows:



3) If C2 and C3 are 0 and 1, then the logic of condition 2 described above will be repeated; however, corresponding status flags will be checked for 0 values as the condition which forces the next microinstruction to be skipped.

4) If C2, C3 and C4 are 1, 1 and 0, respectively, then C5, C6, C7 and C8 specify the status of four control signals which the Control Unit may output at chip pins. We have only described two control signals thus far: READ and WRITE. We will assume that C8 specifies the condition of READ and C7 specifies the condition of WRITE. This use of the nine controls C_0 through C_8 may be illustrated as follows:



5) When C2, C3 and C4 are all 1, then C5 through C8 will be decoded internally to specify one of 16 logical operations internal to the Control Unit. We will not attempt to define what these operations might be.

We will now create some microprograms. Let us begin simply, by creating an instruction fetch microprogram. Recall that every instruction's execution starts with an instruction fetch; therefore, the instruction fetch microprogram must precede every microprogram which implements an instruction's execution. The instruction fetch microprogram is shown in Table 4-4.

Before analyzing the instruction fetch microprogram, microinstruction-by-microinstruction, a few general comments must be made.

MICRO-INSTRUCTION BIT LENGTH

Each microinstruction becomes nine binary digits within the Control Unit. The 8-bit (or byte) unit was selected as the word size for the microcomputer because this word size is useful when representing characters and numeric data, in addition to representing instruction codes. The Control Unit microprogram does not represent numeric data or instruction codes; therefore, the microinstruction bit length is arbitrarily set to whatever the microcomputer needs — in this case nine bits. Since there

Table 4-4. An Instruction Fetch Microprogram

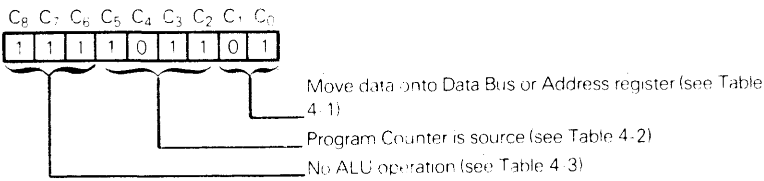
Instruction Number	MICROINSTRUCTION CODE $C_8C_7C_6C_5C_4C_3C_2C_1C_0$	FUNCTION
1	1 1 1 1 0 1 1 0 1	Move Program Counter to Address Register
2	1 0 0 0 0 1 1 1 1	Set READ Control signal true, WRITE false
3	1 1 1 0 0 1 0 0 1	Move Program Counter low order byte to Data Bus
4	1 1 1 1 0 0 1 1 0	Move Data Bus to ALU latches
5	0 1 1 0 0 0 0 0 0	Increment ALU latches
6	1 1 1 1 0 0 1 0 1	Move ALU latches to Data Bus
7	1 1 1 0 0 1 0 1 0	Move Data Bus to Program Counter low order byte
8	1 1 1 1 1 0 0 0 1	Move Program Counter high order byte to Data Bus
9	1 1 1 1 0 0 1 1 0	Move Data Bus to ALU latches
10	1 0 0 0 0 1 0 1 1	Skip next microinstruction if carry status = 0
11	0 1 1 0 0 0 0 0 0	Increment ALU latches
12	1 1 1 1 0 0 1 0 1	Move ALU latches to Data Bus
13	1 1 1 1 1 0 0 1 0	Move Data Bus to Program Counter high order byte
14	1 1 1 1 1 0 1 0 1	Move Data Register to Data Bus
15	1 1 1 1 0 1 0 1 0	Move Data Bus to Instruction Register

are 15 microinstructions in the instruction fetch microprogram, a total of 135 binary digits will be required, in a 9 × 15 binary digit matrix, to hold the instruction fetch microprogram.

Notice also that a sequence of 15 microinstructions are executed during the instruction fetch, which must occur during one period of clock Φ. The Control Unit will therefore internally split the clock signal into 16 subdivisions. In other words, if the clock has a period of one microsecond each microinstruction must execute within 62.5 nanoseconds. Since the average CPU chip consists of densely packed n-MOS or p-MOS logic, this time period is reasonable.

Now consider, in detail, the 15 steps of the instruction fetch microprogram.

The first two bits of the first microinstruction's 9 bit code, representing C0 and C1, are set to and 0, respectively; they indicate that data will be moved onto the Data Bus, or into the Address register (see Table 4-1). The next four bits are set to 1 1 0 1; they specify that it is the contents of the Program Counter which must be moved to the Address register (see Table 4-2). Since no simultaneous ALU operations are to take place, the last three bits are all set to 1 (see Table 4-3). The creation of this microinstruction is illustrated as follows:



If you have any difficulty understanding the creation of the first microinstruction, you should study some of the other microinstructions in detail, to see how they are also created from the information in Tables 4-1, 4-2 and 4-3.

Microinstruction 1 moves the contents of the Program Counter to the Address register, thus making the 16-bit contents of the Program Counter appear at the 16 address pins. Instruction 2 sets the WRITE control signal false and the READ control signal true; this tells external logic that pins A0 through A15 provide the address of an external memory word, the contents of which are to be placed at pins D0 through D7. External logic has 687.5 nanoseconds, that is, the time it takes to execute microinstructions 3 through 13, in which to fetch the requested data.

Microinstructions 3 through 13 increment the contents of the Program Counter, as is required during every instruction fetch. Since the Program Counter is 16 bits long, while logic within the CPU is only 8 bits wide, the Program Counter has to be incremented in two steps. Instructions 3 through 7 increment the low order half of the Program Counter. If this increment results in the carry status being set (in the ALU only), then the high order half of the Program Counter must also be incremented. If the carry status is not set, then the high order half of the Program Counter must remain unaltered. Microinstructions 9 through 13 handle the high order half of the Program Counter. These microinstructions parallel microinstructions 3 through 7; however, microinstruction 10 specifies that if the Carry status (in the ALU) is 0, then microinstruction 11, which actually performs the increment on the contents of the ALU latches, be skipped. Thus the high order half of the Program Counter is only incremented if the Carry status was set when the low order half of the Program Counter got incremented.

STATUS IN MICRO-PROGRAMS

Note that Control Unit logic must be very specific about when it records statuses in its CU DATA buffer and when it does not. Use of the Carry (C) status as a means of controlling the Program Counter increment is only valid if the Carry status is not permanently recorded in the Control Unit. In other words, the Control Unit can reference the status latches in the ALU any time. Assembly language instructions reference the statuses stored in the CU DATA buffer, never the statuses in the ALU latches. Microinstruction code 00000011 must be executed by the Control Unit if the statuses in the ALU latches are to be saved in the CU DATA buffer.

Now consider the five steps needed to complement the contents of the Accumulator. If, during the 15th step of the instruction fetch microprogram, the code loaded into the Instruction register is a Complement Accumulator instruction code, then Control Unit logic will branch to the microprogram shown in Table 4-5.

COMPLEMENT MICRO-PROGRAM

In order to complement the Accumulator, a 45-bit microprogram must be executed. Even though these five microinstructions can be executed in 312.5 nanoseconds, system synchronization demands that one period of clock Φ be set aside for instruction execution; therefore, the remaining time will be wasted.

Let us now consider the tradeoffs associated with having simple or complex instruction sequences. With reference to the binary addition program which was described earlier in this chapter, recall that a word of data can be loaded from memory into the Accumulator in one of the following ways:

ASSEMBLY LANGUAGE INSTRUCTION MICRO-PROGRAMS

1) Issue two separate instructions, each of which loads half of the Data Counter with half of the data memory address for the data memory word whose contents must be loaded into the Accumulator. Then issue a third instruction to load the contents of the addressed data memory word into the Accumulator.

2) Use one instruction to load into the Data Counter the entire data memory address for the word whose contents is to be read into the Accumulator. Then issue a second instruction to move the contents of the addressed data memory word to the Accumulator.

3) Have a single direct addressing instruction which loads the data memory word address into the Data Counter, then loads the contents of the addressed data memory word into the Accumulator.

Table 4-5. A Complement Accumulator Microprogram

Instruction Number	MICROINSTRUCTION CODE $C_8C_7C_6C_5C_4C_3C_2C_1C_0$	FUNCTION
1	1 1 1 0 0 0 0 0 1	Move Accumulator to Data Bus
2	1 1 1 0 0 0 1 1 0	Move Data Bus to Complementer
3	1 0 0 0 0 0 0 0 0	Execute Complementer logic
4	1 1 1 0 0 0 1 0 1	Move Complementer to Data Bus
5	1 1 1 0 0 0 0 1 0	Move Data Bus to Accumulator

The instruction execution phase of each instruction, for the three ways in which data can be loaded into the Accumulator, are shown in Tables 4-6, 4-7 and 4-8.

Table 4-6. Three-Instruction Memory Read

Instruction Number	MICROINSTRUCTION CODE $C_8C_7C_6C_5C_4C_3C_2C_1C_0$	FUNCTION
1		Repeat microinstructions 1 through 14 of Instruction Fetch (Table 4-4)
14		
15	1 1 1 0 1 0 0 1 0	Move Data Bus to Data Counter, low order byte

(A) Load low order half of Data Counter

Instruction Number	MICROINSTRUCTION CODE $C_8C_7C_6C_5C_4C_3C_2C_1C_0$	FUNCTION
1		Repeat microinstructions 1 through 14 of Instruction Fetch (Table 4-4)
14		
15	1 1 1 1 0 0 0 1 0	Move Data Bus to Data Counter, high order byte

(B) Load high order half of Data Counter

Instruction Number	MICROINSTRUCTION CODE $C_8C_7C_6C_5C_4C_3C_2C_1C_0$	FUNCTION
1	1 1 1 0 0 1 1 0 1	Move Data Counter to Address Register
2	1 0 0 0 1 0 1 1 1	Set READ Control signal true, WRITE false
3	1 1 1 0 0 0 0 0 0	Include 12 no operations to give external logic more time to fetch data
14	1 1 1 0 0 0 0 0 0	
15	1 1 1 1 1 0 1 0 1	Move Data Register to Data Bus
16	1 1 1 0 0 0 0 1 0	Move Data Bus to Accumulator

(C) Load Addressed Data Memory Word Contents Into Accumulator

Table 4-7. One Instruction To Load 16-Bit Address Into Data Counter

Instruction Number	MICROINSTRUCTION CODE $C_8C_7C_6C_5C_4C_3C_2C_1C_0$	FUNCTION
1 — 14		Repeat microinstrucions 1 through 14 of Instruction Fetch (Table 4-4)
15	1 1 1 0 1 0 0 1 0	Move Data Bus to Data Counter, low order byte
16	1 1 1 0 0 0 0 0 0	Timing filler
17 — 30		Repeat microinstructions 1 through 14 of Instruction Fetch (Table 4-4)
31	1 1 1 1 0 0 0 1 0	Move Data Bus to Data Counter, high order byte

Table 4-6(C) provides second step for Table 4-7.

Table 4-8. Single Instruction, Direct Addressing, Memory Read

Instruction Number	MICROINSTRUCTION CODE $C_8C_7C_6C_5C_4C_3C_2C_1C_0$	FUNCTION
1 — 14		Repeat microinstructions 1 through 14 of Instruction Fetch (Table 4-4)
15	1 1 1 0 1 0 0 1 0	Move Data Bus to Data Counter, low order byte
16	1 1 1 0 0 0 0 0 0	Timing filler
17 — 30		Repeat microinstructions 1 through 14 of Instruction Fetch (Table 4-4)
31	1 1 1 1 0 0 0 1 0	Move Data Bus to Data Counter, high order byte
32	1 1 1 0 0 0 0 0 0	Timing filler
33	1 1 1 0 0 1 1 0 1	Move Data Counter to Address Register
34	1 0 0 0 1 0 1 1 1	Set READ control signal true, WRITE false
35 — 46	1 1 1 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0	Include 12 no operations to give external logic more time to fetch data
47	1 1 1 1 1 0 1 0 1	Move Data Register to Data Bus
48	1 1 1 0 0 0 0 1 0	Move Data Bus to Accumulator

The briefest glance at Tables 4-6, 4-7 and 4-8 shows that microprograms are going to have a lot of duplicated microinstruction sequences. The very first thing a microcomputer designer will do is try to eliminate this duplication by re-using frequently needed microinstruction sequences.

Also, a microcomputer designer is going to develop some simple means of giving external logic time to respond to a READ request, rather than having 12 no operation microinstructions, using up 108 bits of the Control Unit storage space.

These are the complications which forced the early microcomputer designers to keep microcomputer assembly language instructions simple. There are many ways in which microprogram sequences can be re-used, and time delays can be implemented; we left 16 microinstructions free for just this kind of operation. Precious Control Unit storage is used up solving these complications, and the more complications there are within a single instruction, the more complex this extra Control Unit logic gets to be.

CHIP SLICE BASED MICROCOMPUTERS

Suppose a microprocessor based microcomputer will not meet your needs; frequently this will happen because the microprocessor is not fast enough. **You are now a candidate for "chip slice" or "macro logic" based microcomputers,** which let you design and build your own CPU, with any CPU architecture (within limits), and any, or no assembly language instruction set.

Before we examine what a chip slice product must consist of, a word of caution. This discussion of chip slice products is something of a tangent within the context of products discussed in this book.

Up to this point we have been describing microprocessors — CPU logic that will be implemented on a single chip, or may be part of a single chip. Chapter 5 describes additional logic which supports microprocessor based microcomputers.

As compared to microprocessors, chip slice products take a wholly different philosophical direction:

We build chips with less logic, where each chip provides one of the fundamental building blocks of any CPU; then we justify less logic — and therefore more chips — with increased performance.

Thus, you could use chip slices to build the equivalent of any microprocessor. You would then have a product with perhaps ten chips instead of one, but it would execute instructions ten times as fast.

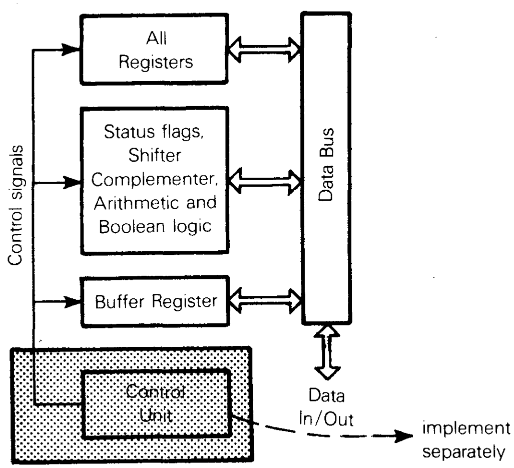
Describing chip slices at the end of Chapter 4 implies that chip slices are essentially CPU building blocks.

That is the frame of reference in which we choose to describe chip slices; however, when you have finished reading Chapter 5, you will realize that chip slices could be used equally well to build the equivalent of any support logic device, excluding ROM or RAM.

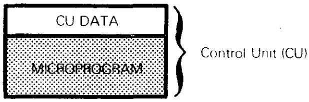
CHIP SLICING PHILOSOPHY

If we are to create CPU building blocks, how should CPU logic be divided so that the resulting pieces are very general purpose?

We cannot impose instruction set limitations; if we do, the CPU building blocks will not be general purpose. Therefore, we begin by separating control unit logic from the rest of the CPU:



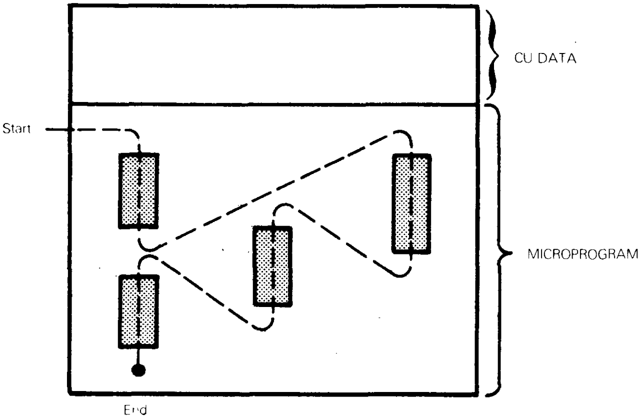
Now if you refer back to our discussion of microprogramming, you will see that a Control Unit, in reality, consists of a microprogram stored in Read Only Memory:



The shaded area marked "microprogram" contains microinstruction sequences just like the sequences illustrated in Tables 4-5 through 4-8.

CU DATA represents a small read/write memory work space needed by the Control Unit.

What we have ignored, so far, is the logic which will allow you to pick your way around the microprogram ROM —concatenating short microinstruction sequences into any macroinstruction's response microprogram:

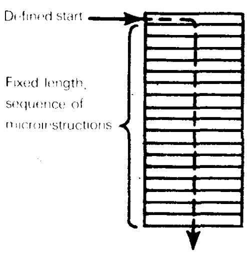


The illustration above arbitrarily shows four separate microinstruction sequences (shaded), which must be executed in order to enable the logical sequence of events required by some undefined macroinstruction. The broken line identifies the order in which macroinstruction sequences must be executed.

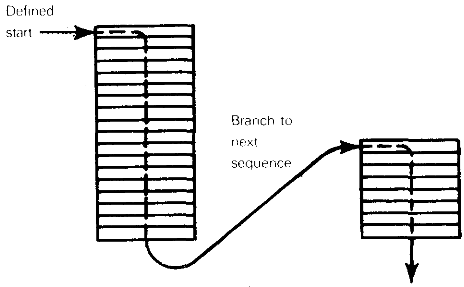
MICRO-PROGRAM SEQUENCER LOGIC

Our Control Unit must have microprogram sequencer logic which allows it to pick its way around the microprogram ROM, as illustrated above by the broken line. Let us look at some of the functions that our Microprogram Sequencer Logic must be able to perform:

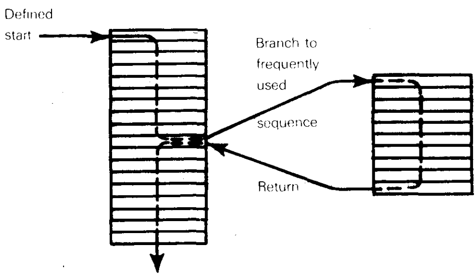
1) It must access a contiguous sequence of microinstructions, beginning with a defined first microinstruction, and continuing for a fixed number of microinstructions:



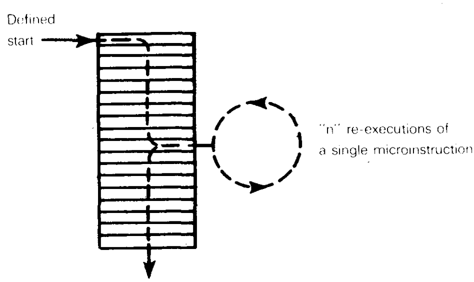
2) It must be able to branch to another contiguous microinstruction sequence:



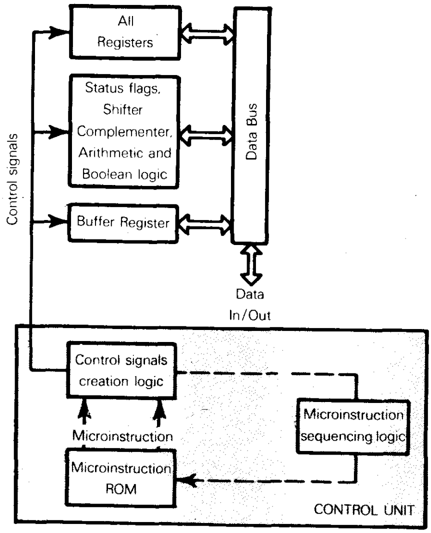
3) It must be able to branch to a frequently used microinstruction sequence, such as a memory access, then return to the point from which it branched:



4) It must be able to continuously re-execute a single microinstruction, such as a No Operation, some fixed number of times:

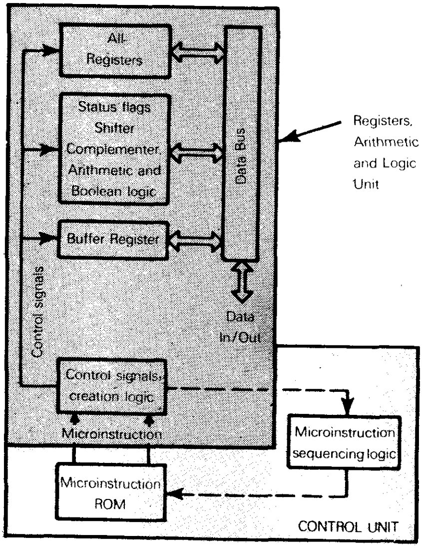


The Control Unit, in reality, will become a microprogram ROM and associated microinstruction sequencing logic:



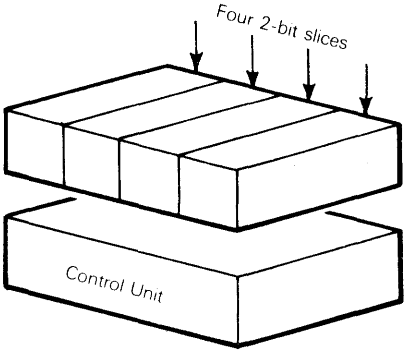
REGISTERS, ARITHMETIC AND LOGIC UNIT CHIP SLICE

Now in practice, it is easier to implement Control signals creation logic as part of the Registers, Arithmetic and Logic Unit:

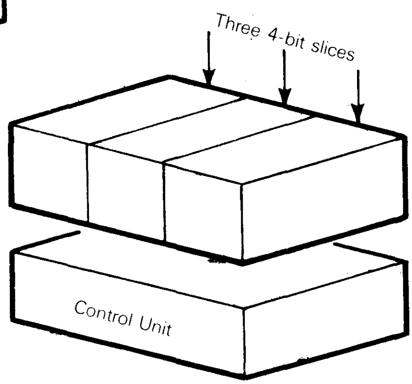


We will begin our discussion of chip slice products with the Registers, Arithmetic and Logic Unit, which we will slice up into segments.

In dividing up this logic, it is imperative that we place as few restrictions as possible on the number and organization of registers. Also, we cannot limit CPU word size; even though we talk consistently about 8-bit microcomputers, it would be very short sighted to assume that an 8-bit word size is going to last forever. Within the microcomputer industry you measure "ever" in months, not years. **We will therefore slice up our registers, arithmetic and logic unit into identical vertical slices, such that slices may be stacked to form a CPU with any word size:**



An 8-bit CPU



A 12-bit CPU

We will refer to each slice as an ALU slice.

ALU SLICE

2-bit ALU slices and 4-bit ALU slices are commonly seen. So long as your word size is a multiple of 4, the 4-bit slice is superior, since it requires fewer chips.

If the combined registers and ALU logic is to be sliced up, each slice must be able to interface with an identical neighbor on either side, in addition to a Control Unit.

Any simple ALU organization, such as illustrated in Figure 4-1, presents a lot of problems. The innumerable data paths converging on the Data Bus are going to become even more complex, since the registers, if they are to be general purpose, cannot be predefined or limited in number, as shown. You would have to construct impractical microinstructions to identify the innumerable valid data path combinations. Therefore we will reorganize our registers and ALU with an eye to streamlining the data paths, while maintaining flexibility. Remember, a successful chip slice makes no assumptions regarding the architecture of the end product.

Now there are a very large number of microprocessor-based microcomputers on the market and the number constantly increases; but there are very few chip slice products. Therefore **we will**

reorganize the registers and ALU portion of Figure 4-1 to generally conform with the organization of 2900 and 6700 series 4-bit chip slice products which are described in Volume II. Figure 4-3 illustrates this reorganization. 3000 series 2-bit chip slice products are conceptually very similar, and represent the predecessor of 2900 and 6700 series products. 10800 chip slice products represent the next generation — the logical evolution of 2900 and 6700 series products.

Figure 4-4 illustrates the concept of a "chip slice"; the figure shows two 4-bit slices creating an 8-bit ALU.

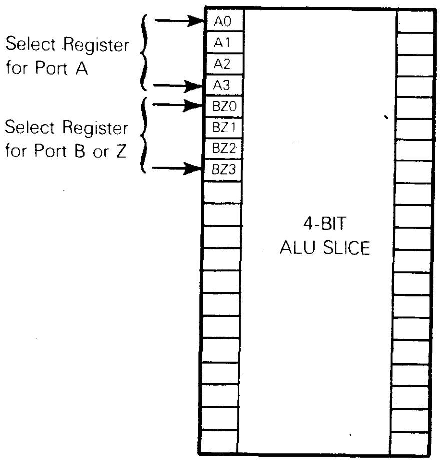
Let us look at Figure 4-3 in overview.

REGISTER BLOCK

Some fixed number of registers must be specified **within the Registers block. 16 registers are selected,** since a 4-bit select code can address one of the 16 registers.

The Registers block has two output ports, AA and BB, plus one input port, ZZ. Having three ports, the register block needs three sets of register select logic — one for each port. Select logic identifies the register which is effectively connected to each port at any point in time. We can get by with two sets of register select logic by combining input ZZ select logic with either output port AA or BB select logic. We will arbitrarily choose to combine ZZ and BB select logic. This means that at any point in time the same register in the register block will be effectively connected to both the ZZ and BB ports.

Thus **the chip slice DIP needs four A port register select pins and four B port register select pins:**



DATA PATHS

Now consider the data paths between the Register's block and the ALU block.

The ALU block requires two input ports, marked PP and QQ, since a number of ALU operations require two inputs to create one output. RR marks the output. Input port PP can receive Registers block output AA or BB, or it can receive contents of the Buffer register. XX marks a three-way junction whence input PP is derived.

Input port QQ can receive either Registers block BB output, or external data. YY marks the two-way junction whence the QQ input is derived.

Let us examine the external signals that will be required to support the interface between the register and ALU blocks. If we assume that an additional option is to input 0 at ports PP or QQ then **the following input combinations are allowed:**

ALU INPUT IDENTIFIED

QQ:	0	0	0	0	BB	BB	BB	BB	DI	DI	DI	DI
PP:	0	AA	BB	VV	0	AA	BB	VV	0	AA	BB	VV

BB - 0 is the same as 0 - BB, so ignore it.

Since AA and BB can have the same value, ignore BB - BB which can be made equivalent to BB - AA; ignore DI -BB which can be made equivalent to DI -AA.

0 - 0 is ignored since an ALU operation will never require two 0 inputs.

We will use three input pins to identify the remaining eight possible PP-QQ input combinations. These three input pins will become the low order three bits of a 9-bit microinstruction code and will be interpreted as shown in Table 4-9.

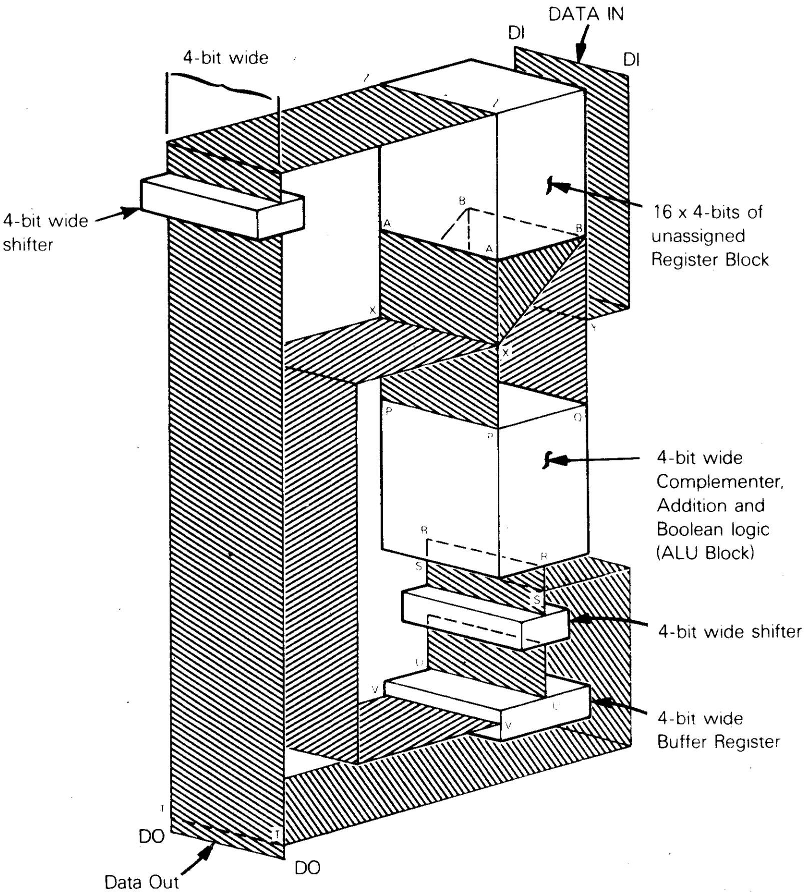


Figure 4-3. Register, Arithmetic And Logic Unit From Figure 4-1, Reorganized To Meet The Needs Of A Chip Slice

Table 4-9. ALU Sources As Defined By The Low Order Three Microinstruction Bits

MICROINSTRUCTION			ALU INPUTS	
I2	**I1**	**I0**	**QQ**	**PP**
0	0	0	BB	VV
0	0	1	BB	AA
0	1	0	00	VV
0	1	1	00	AA
1	0	0	00	BB
1	0	1	DD	BB
1	1	0	DD	VV
1	1	1	DD	00

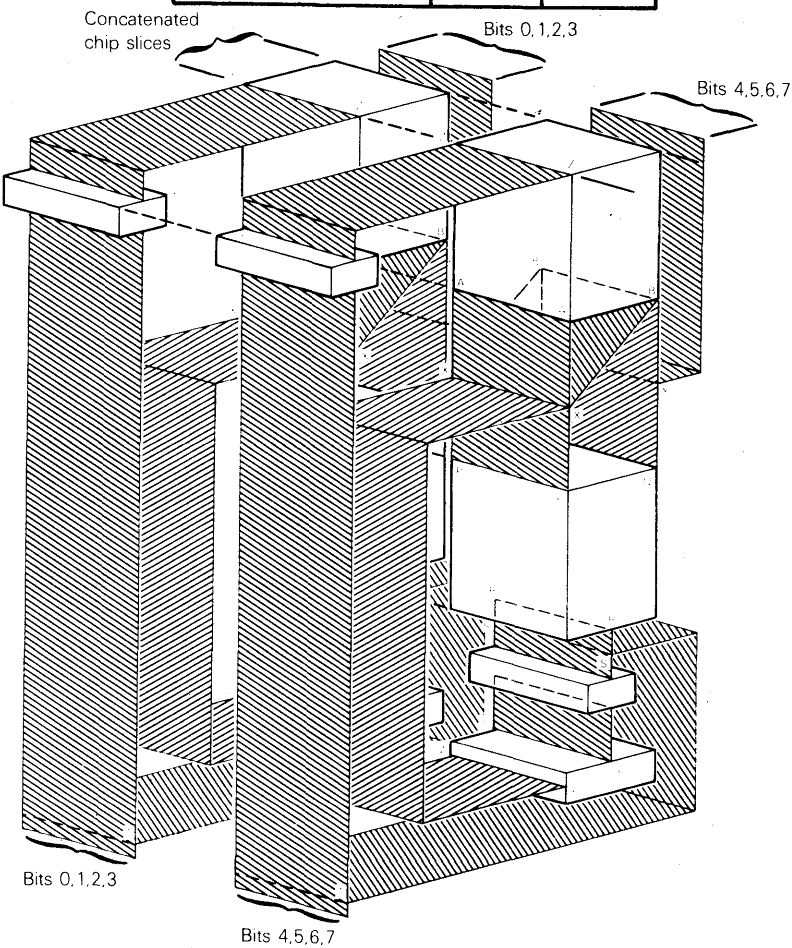
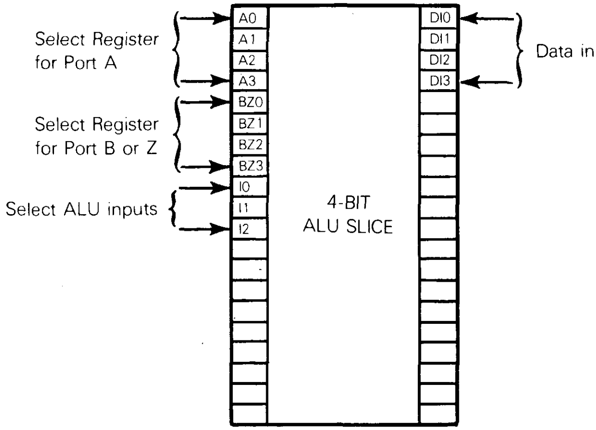


Figure 4-4. Two 4-Bit ALU Slices Concatenated To Generate An 8-Bit ALU

The Registers block-ALU interface also requires four data in pins supporting the data input to YY. Our DIP therefore looks like this:



CHIP SLICE ARITHMETIC AND LOGIC UNIT

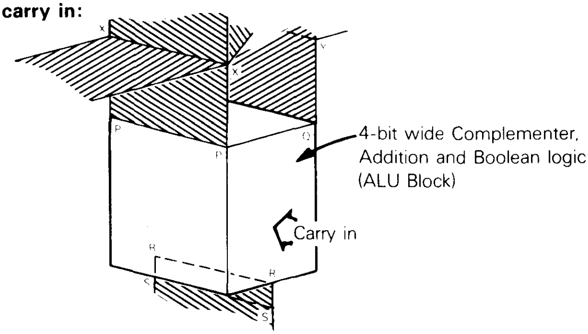
Now move on to the Arithmetic and Logic Unit.

Notice that the shifter has been moved out, leaving behind complementer, addition and Boolean logic. Moving shifter logic out allows the option of shifting data within a short recycle path through the Buffer register (RRSSUUVVXXPP) or we can shift a final ALU operation result on its way back to the Registers block (RRSSTTZZ).

The Buffer register in Figure 4-3 does not serve the same purpose that it did in Figure 4-1. In Figure 4-1 the Buffer register provides the second ALU input whenever an ALU operation requires two inputs. In Figure 4-3 the ALU inputs come from the two Registers block output ports PP and QQ. In Figure 4-3, the Buffer register has become a holding location for intermediate results of ALU operations.

CHIP SLICE ALU OPERATION IDENTIFICATION

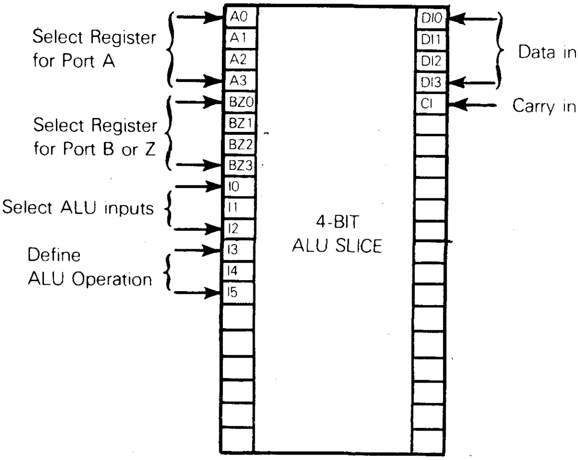
We will assign the next three bits of the microinstruction code (I3, I4, I5) to define the ALU operations which are to be performed. There are only five isolated operations: ADD, COMPLEMENT, AND, OR, XOR. We could add increment and decrement to the list; instead we generate the equivalent by providing an external carry in:



Output RR results from inputs PP, QQ and Carry in.

Combining the two input options with the carry in and the five ALU operations allows us to generate the ALU operation codes illustrated in Table 4-10.

Our 4-bit ALU slice DIP will need three more microinstruction inputs, plus a Carry in:



Only the ALU destination remains to be specified; we will use three microcode bits for this specification.

CHIP SLICE ALU DESTINATION

These are the three possible destinations for ALU block output:

1) The Buffer register, via SS and UU.
2) The Registers block, via SS, TT and ZZ.
3) Data out, via SS and TT.

Data on its way to the Buffer register or the Registers block may optionally be shifted left or right A bewildering variety of output options could be selected since data can be output to any or all of

Table 4-10. ALU Operations Specified By Middle Three Microcode Bits

MICROCODE	FUNCTION		
I5 I4 I3	**General**	**Carry In = 0**	**Carry In = 1**
0 0 0	QQ + PP	QQ + PP PP if QQ is 0 QQ if PP is 0	QQ + PP + 1 Increment PP if QQ = 0 Increment DD if PP = 0
0 0 1	QQ + ($\overline{PP}$) ($\overline{PP}$ is the ones complement of PP)	QQ - PP - 1 Ones complement PP if QQ is 0	PP - QQ Twos complement PP if QQ is 0
0 1 0	QQ OR PP	Carry in plays no part in Boolean operations	
0 1 1	QQ AND PP		
1 0 0	QQ XOR PP		
1 0 1	currently unassigned		
1 1 0	currently unassigned		
1 1 1	currently unassigned		

three destinations, with shifting occuring along two of the destination paths. Until you have used a chip slice product quite extensively, it will not be clear which of the output path options are useful.

Table 4-11. ALU Destinations Specified By
Last Three Microcode Bits

I8 I7 I6	BUFFER REGISTER		REGISTER BLOCK		Data Out
	Shift		Shift		
0 0 0	No	Yes		No	Yes
0 0 1	Left	Yes		No	Yes
0 1 0	Right	Yes		No	Yes
0 1 1		No		No	Yes
1 0 0		No	No	Yes	Yes
1 0 1		No	Left	Yes	Yes
1 1 0		No	Right	Yes	Yes
1 1 1	No	Yes	No	Yes	Yes

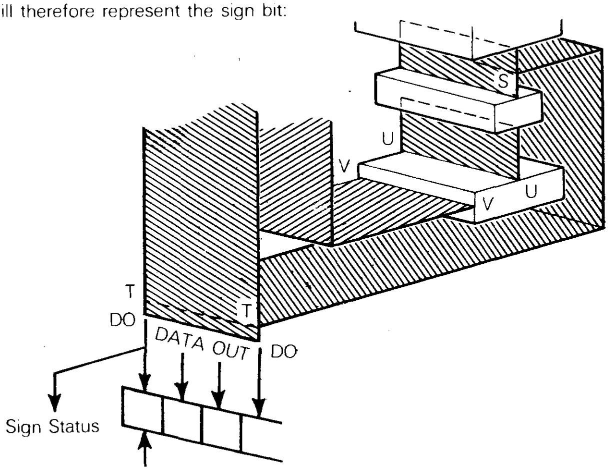
Bear in mind that **we really have two types of ALU output — the temporary data which is heading for the Buffer register and permanent answers which are heading back to the Registers block. Based on this concept, Table 4-11 illustrates one way in which destinations could be specified.**

CHIP SLICE STATUS

The only subject left to discuss is status. The Zero, Overflow and Sign status flags are easy to generate, so let us look at these three first.

SIGN STATUS

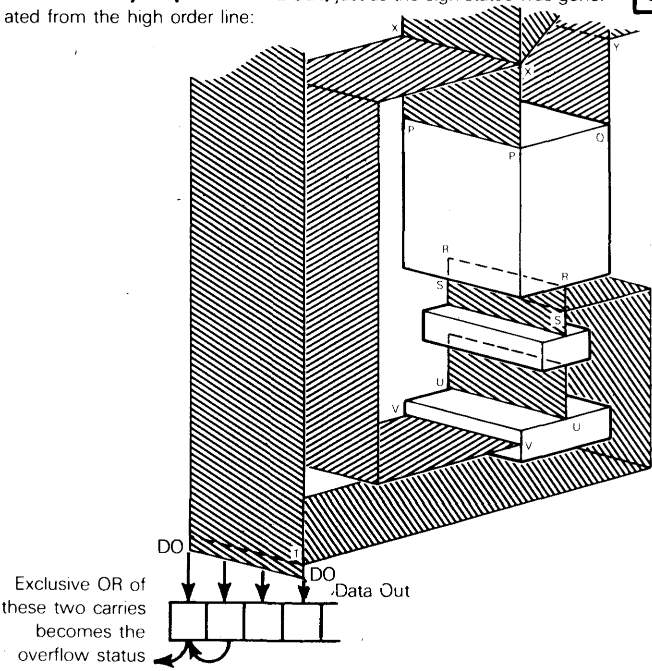
Every chip slice will be built assuming that it can be the high order slice in the ALU. Every slice will therefore have logic which assumes that the data out lines represent the four high order bits of the eventual ALU word. The high order line will therefore represent the sign bit:



We can generate the sign bit directly from the high order data out line of every single chip slice. Only the high order chip slice's Sign bit will be used; other chip slice sign bits will be ignored.

The overflow status can be generated from the two high order lines of every chip slice data out, just as the sign status was generated from the high order line:

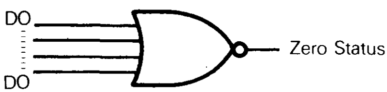
OVERFLOW STATUS



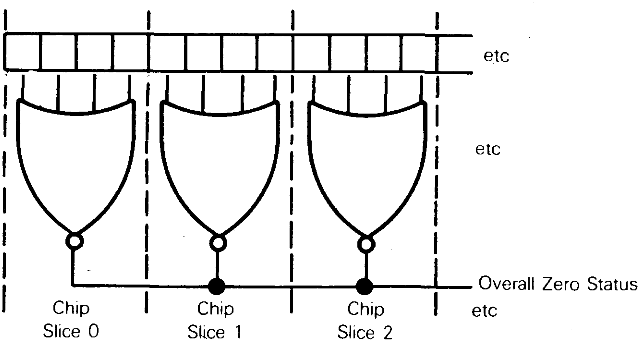
As described in Chapter 2, the overflow status represents the Exclusive OR of carries out of the penultimate and ultimate bits of a data word. OVERFLOW LOGIC CAN THEREFORE ONLY BE GENERATED WITHIN THE ALU.

Generating a zero status is also quite straightforward. For every chip slice we will output NOT OR of the four data out lines:

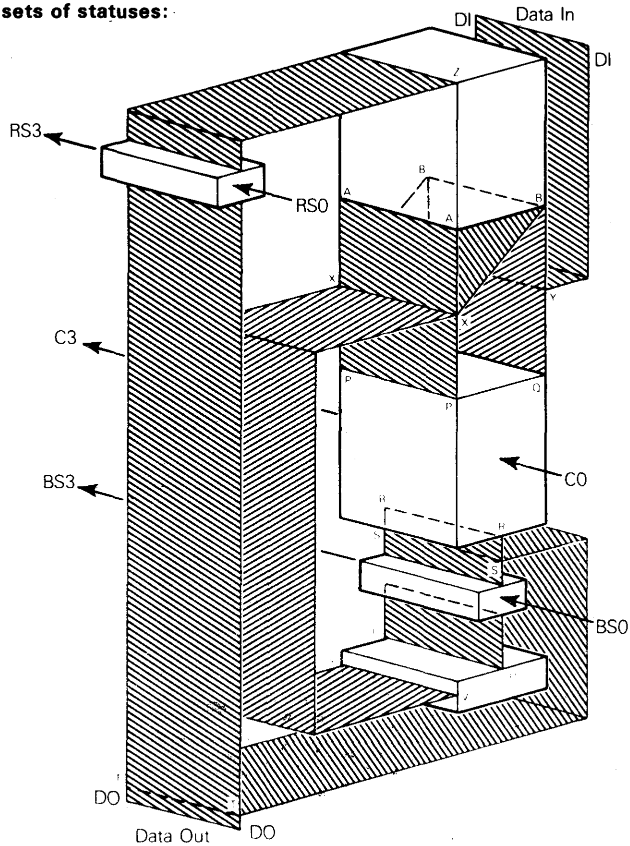
ZERO STATUS



By tying the Zero statuses of all chip slices within the CPU together you can create an overall Zero status:

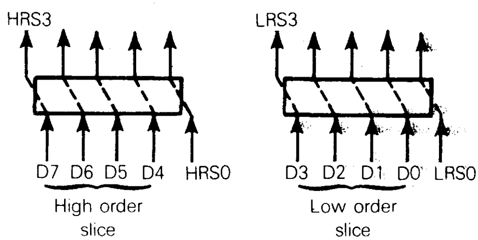


Carry and shift statuses are not nearly so straightforward. First of all, we need three sets of statuses:



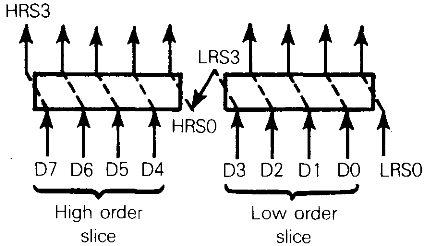
But the statuses illustrated above will not always work efficiently, since chip slices must work in parallel.

Consider a simple, 8-bit shift, created using two chip slices:



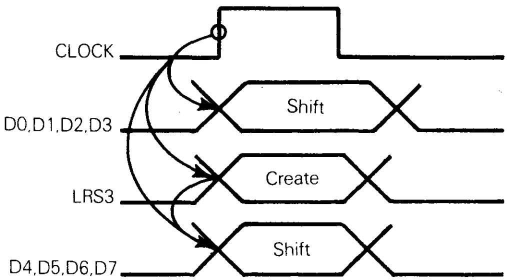
If the shift illustrated above is to occur as a single, parallel step, the low order slice shift out (LRS3) must become the high order slice shift in (HRS0).

If LRS3 and HRS0 are both connected to DIP pins, all we have to do is connect these two pins and an 8-bit shift is created:

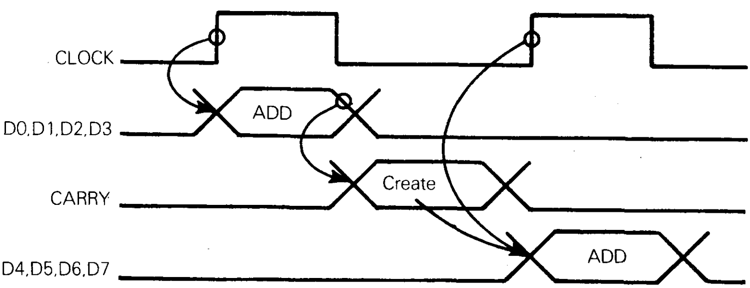


But when it comes to binary addition, the problem is less straightforward. When shifting, LRS3 is created while the shift is in progress. When adding, the carry out is generated at the end of the addition.

Here is a simplified illustration of this timing problem. For a shift, we have no problem:

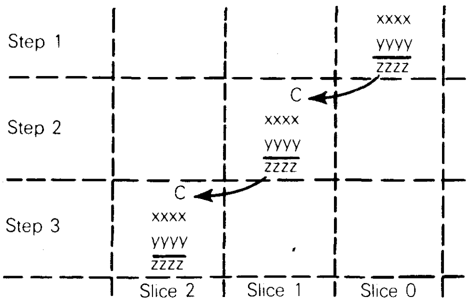


For binary addition, we have a problem:



CARRY STATUS

We could perform the binary addition in four-bit increments, starting with the least significant four bits; and in this case, carry could simply be rippled from one 4-bit slice to the next:



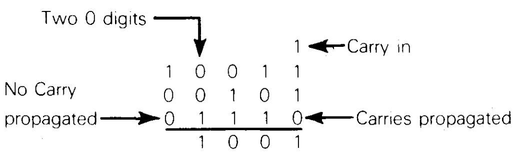
CARRY LOOK AHEAD

But **rippling binary addition defeats the whole purpose of using chip slice products** — to gain instruction execution speed. We must therefore add logic which allows the ALU to forecast whether a binary addition is going to create a carry, or propagate one coming in.

The rules for carry creation and propagation are simple enough.

CARRY PROPAGATION

First consider carry propagation. If there is a carry in to binary addition, then there will be a carry out so long as no two 0 digits are being added — and thus breaking the propagation chain:

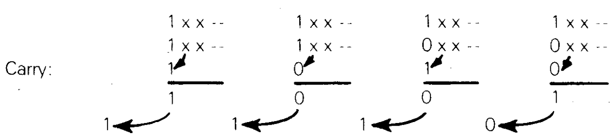


If P_i and Q_i represents binary digits entering the ALU via ports PP and QQ respectively, we conclude that a carry will propagate if:

$(P_0 \text{ OR } Q_0) \text{ AND } (P_1 \text{ OR } Q_1) \text{ AND } (P_2 \text{ OR } Q_2) \text{ AND } (P_3 \text{ OR } Q_3) = 1$

CARRY GENERATION

In order to determine whether a new carry will be generated, we must start at the high order end of the 4-bit unit and work back to the low order end. Both high order digits must be one or one high order digit must be 1 with a carry propagated from the penultimate digits:



If C_i represents the carry out of bit position i, then if $C_i = 1$, generating a carry,

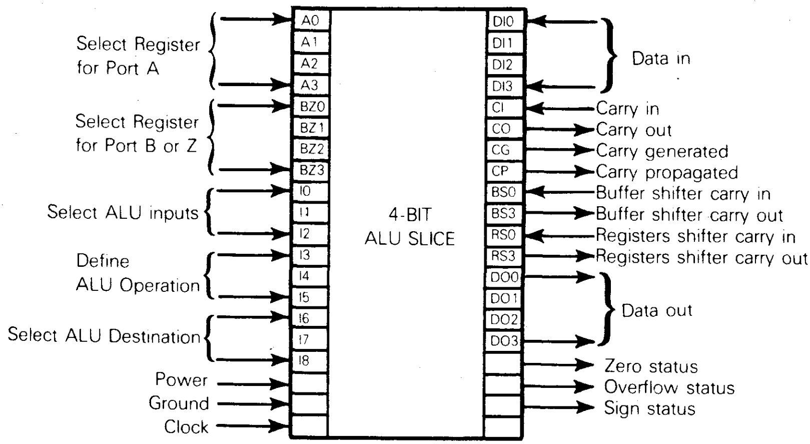
$(P_3 \text{ AND } Q_3) \text{ OR } (C_2 \text{ AND } (P_2 \text{ OR } Q_2)) = 1$

For C_2 to = 1, the same relationship applies, with bit positions shifted down:

$(P_2 \text{ AND } Q_2) \text{ OR } (C_1 \text{ AND } (P_1 \text{ OR } Q_1)) = 1$

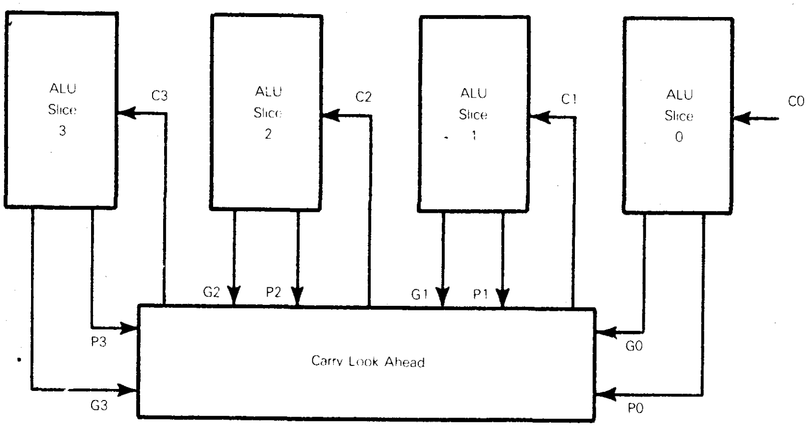
In this fashion the ALU can be provided with logic that predicts a carry generation.

Finally, this is the way pins must be assigned:



CARRY GENERATE DEVICE

Carry generate logic will usually be provided on a separate carry generate device. This device receives Carry Propagate (P) and Carry Generate (G) signals, in the proper sequence, from the 4-bit ALU slices; it generates and returns the correct Carry in (C) to each chip slice:



THE CHIP SLICE CONTROL UNIT

The ALU slices, as we have described them, are driven by a 9-bit microinstruction, together with binary data input and various status/control signals.

The control unit must provide the 9-bit microinstruction code; it could also provide the input status/control signals but typically it does not, for reasons we will soon discuss.

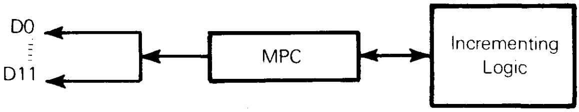
We are going to store the microinstruction code in a very fast Read Only Memory and create addressing logic which accesses microinstructions in the proper sequence. **The Control Unit then consists of the microinstructions ROM and its addressing logic, as discussed earlier in this chapter.**

We can gain a lot of insight into desirable Control Unit addressing logic features by looking back

at the microprocessor microinstruction sequences which were developed in Tables 4-5 through 4-8.

MICRO-PROGRAM COUNTER

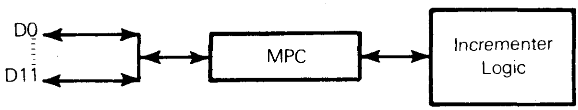
Under normal circumstances microinstruction codes are accessed sequentially. Therefore, **the Control Unit addressing logic must have a Microprogram Counter (MPC), the equivalent of a Program Counter** which can be incremented after every microprogram access to reference the next sequential microinstruction:



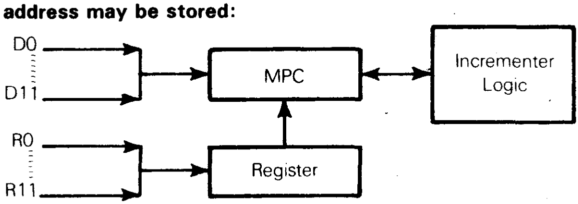
We arbitrarily assume a 12-bit width for the Control Unit address logic — implying a maximum of 4096_{10} microinstructions in the ROM.

Any microinstruction sequence is going to begin at some initial address; therefore **Control Unit addressing logic must be able to initialize the Microprogram Counter.** Consider two possibilities:

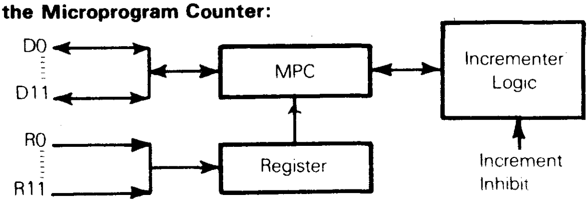
1) Every macroinstruction object code is going to be implemented by a microinstruction sequence with its own initial address which must be loaded into the Microprogram Counter. **We will therefore provide direct data access to the Microprogram Counter:**



2) It would be highly desirable to have some general purpose microprogram origins to handle special circumstances, or alarm conditions that may have nothing to do with execution of an individual instruction. **We will therefore provide a register where some** such **permanent address may be stored:**

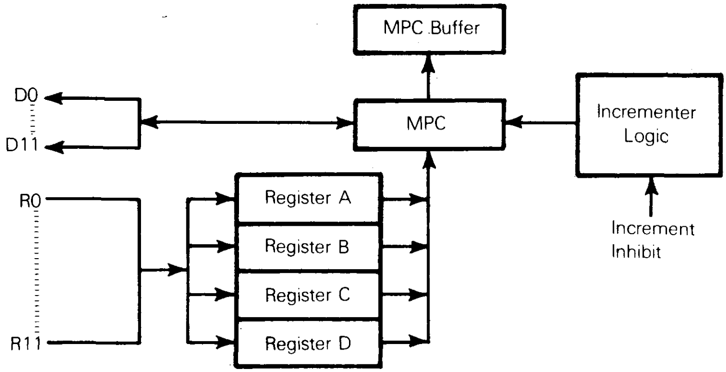


Recall that the Control Unit addressing logic must be able to re-execute one instruction a number of times. In our example, a "No Operation" instruction was re-executed simply to keep the Control Unit synchronized with external timing. **We will therefore add an increment inhibit control to the Microprogram Counter:**

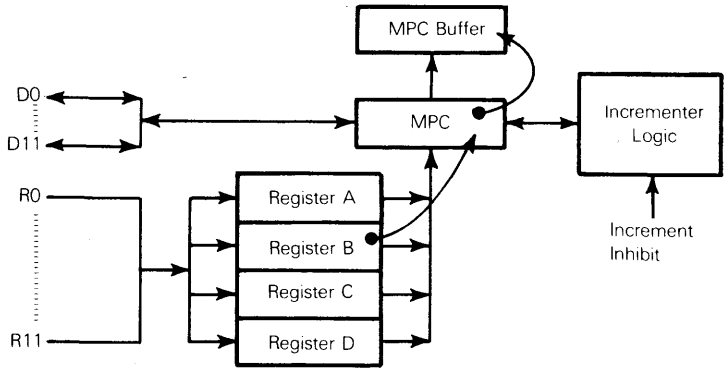


Finally, **recall that there are frequently used microinstruction sequences which perform operations such as memory read or memory write.** We can handle this situation in one of two ways.

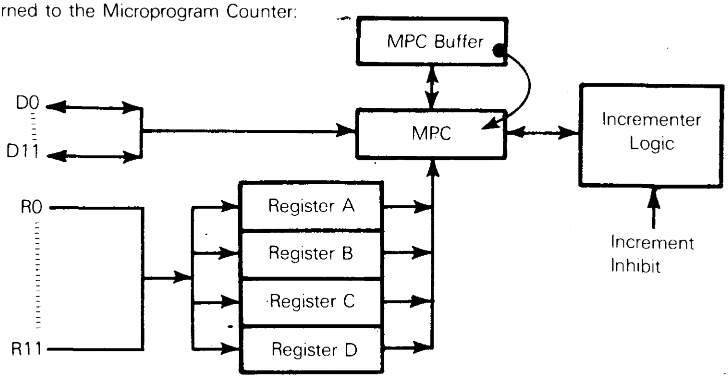
First consider having a number of address registers plus a Microprogram Counter Buffer:



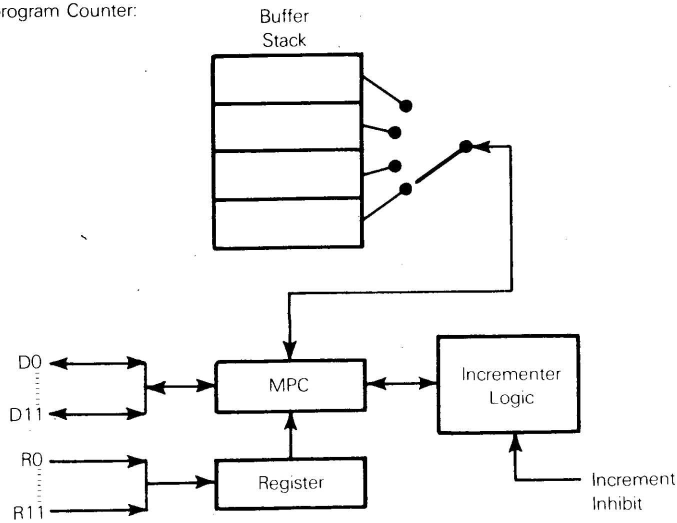
As illustrated above, the address of the first microinstruction for four frequently used microinstruction sequences may be stored in Registers A, B, C and D. The Control Unit addressing logic can save prior contents of MPC in the Buffer, then load the contents of one register:



The last microinstruction in the frequently used sequence causes the Buffer contents to be returned to the Microprogram Counter:



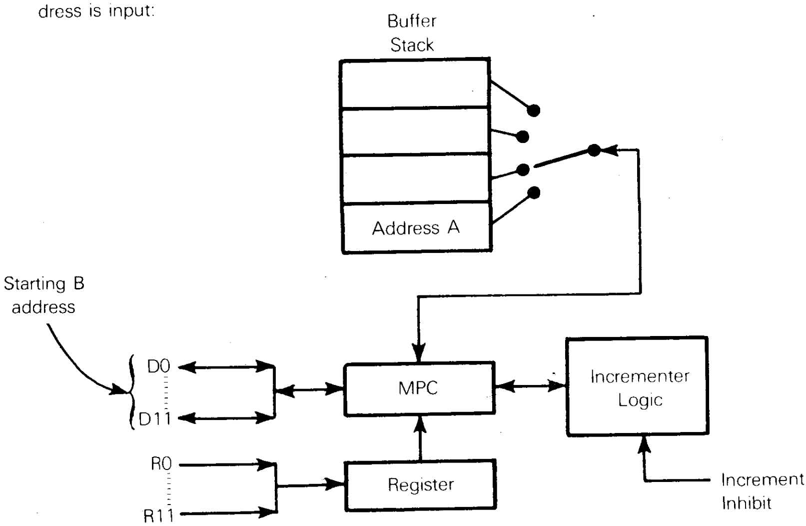
Another approach requires external logic to provide the starting address of each frequently used microinstruction sequence. In this case, **a stack of Buffer registers** will back up the Microprogram Counter:



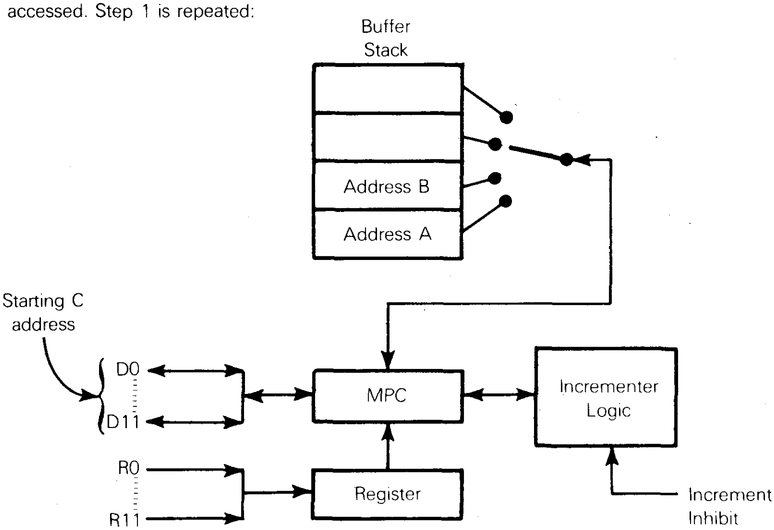
The buffer stack allows one frequently used microinstruction sequence to access another frequently used microinstruction sequence. This is sequence nesting. The stack is a common microcomputer feature and is described in Chapter 6.

This is how the Buffer stack would work for microinstruction sequence A accessing microinstruction sequence B, which in turn, accesses microinstruction sequence C:

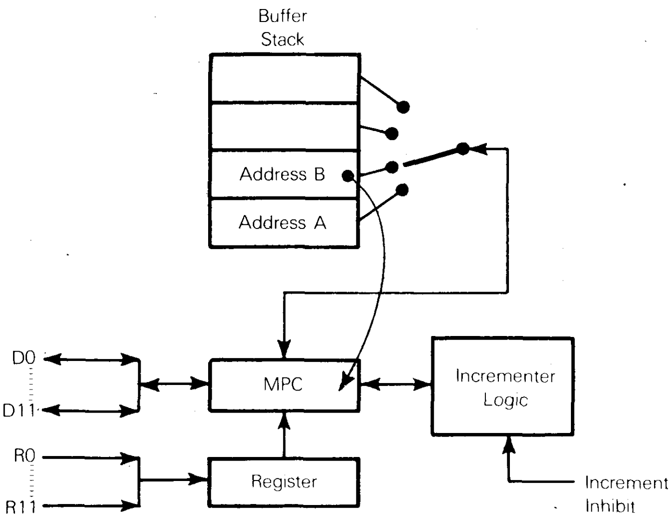
1) Microinstruction sequence A reaches the point where microinstruction sequence B must be accessed. The current sequence A address is saved on the stack, then the sequence B address is input:



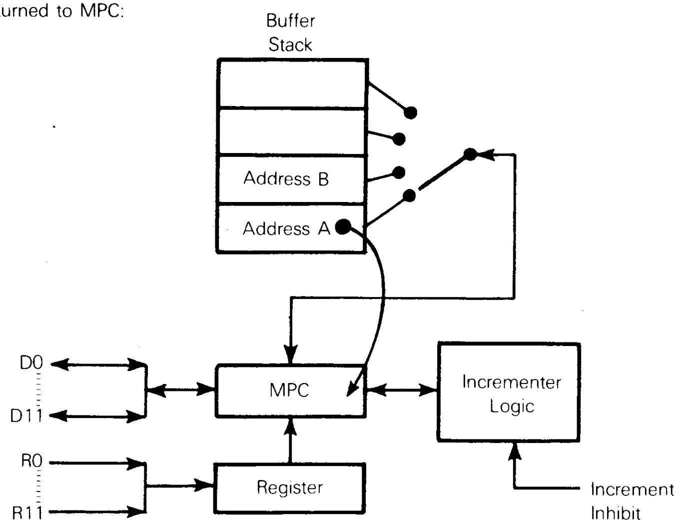
2) Microinstruction sequence B reaches the point where microinstruction sequence C must be accessed. Step 1 is repeated:



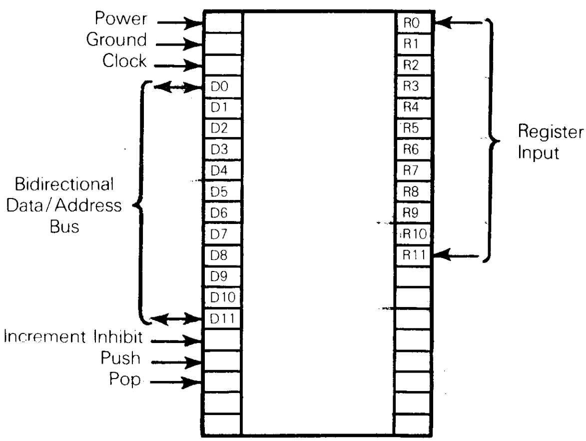
3) Microinstruction sequence C completes execution, so the saved Address B is returned to MPC:



4) Microinstruction sequence B, in turn, completes execution, so the saved Address A is returned to MPC:

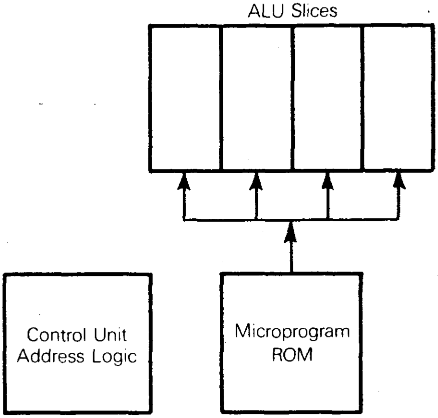


Assuming that our Control Unit addressing logic has a Buffer stack, **two additional control signals will be needed:** one will push the contents of MPC into the stack, as illustrated in Steps 1 and 2; the other will pop the top stack address into MPC, as illustrated in Steps 3 and 4. **Our Control Unit address logic DIP pin assignments will now be as follows:**



COMBINING ARITHMETIC AND LOGIC UNIT WITH CONTROL UNIT

Conceptually **we are going to build a Central Processing Unit by combining ALU slices with Control Unit addressing logic and a microprogram in Read Only Memory** as follows:



In practice a very considerable amount of additional external logic will be required before the simple configuration illustrated above can perform as a Central Processing Unit. For example, nowhere have we addressed the problem of receiving or transmitting control signals. What about the microprocessor Read and Write control signals?

It would have been possible to add logic to the Control Unit that automatically senses and creates CPU-type control signals. However, that assumes chip slice products are going to be used as CPU building blocks only. The assumption is unwarranted.

By describing chip slice products in Chapter 4, a chapter devoted to Central Processing Units, we cast chip slice products as CPU building blocks, which makes them conceptually easy to understand only because of the sequence in which information is being presented in this book.

Excluding control signal processing logic from the chip slice Control Unit means a lot of extra work and extra logic must surround the chip slice and Control Unit set. But, at the same time, no restrictions are imposed on the way these products are used.

In terms of our current discussion, therefore, we must conclude without demonstrating the equivalent of an instruction fetch or a typical instruction's execution, because the type of information which would have to be covered before necessary external logic could be adequately treated is beyond the scope of this book.

Chapter 5
LOGIC BEYOND THE CPU

In this chapter we are going to identify the additional logic that must accompany a CPU in order to generate a microcomputer system that is comprehensive enough to be useful.

We must separately identify the logical components of a microcomputer system by function (e.g., CPU, RAM memory, ROM memory, etc.) but there is no fundamentally necessary correlation between logical components and individual chips. As you will see in Volume II, there can be wide variations between the type of logic which one microcomputer manufacturer will put on a single chip, as compared to another.

PROGRAM AND DATA MEMORY

External memory is the first and most obvious addition needed to support the CPU that was described in Chapter 4.

READ-ONLY MEMORY (ROM)

Interfacing ROM to a CPU is very simple.

As described in Chapter 3, entire words of memory are implemented on a single ROM chip. By contrast, read-write memory (RAM) may require a separate chip for every bit of the memory word.

The signals required by a ROM device are quite elementary. As one would logically expect, a ROM device will require the following input signals:

1) The address of the memory word being accessed.
2) A read control signal which tells the ROM device when to return the contents of the addressed memory word.
3) A clock signal so that the ROM and CPU will be synchronized.
4) Power and ground.

The only output signals which the ROM device must have are eight data lines (for an 8-bit word), via which the contents of the addressed memory word may be transferred back to the CPU. Figure 5-1 illustrates a hypothetical ROM device. This ROM device is connected to the CPU as illustrated in Figure 5-2.

Before describing a ROM access in detail, let us consider some of the non-obvious features of a microcomputer system.

EXTERNAL DATA BUS

Were a microcomputer system to consist of just the CPU and one ROM, pins from the two devices could be directly connected. Since it must be possible for more than two devices to be present in a microcomputer system, signal connections are made via an External Data Bus, which may be likened to a common signal highway connecting chips of the microcomputer set.

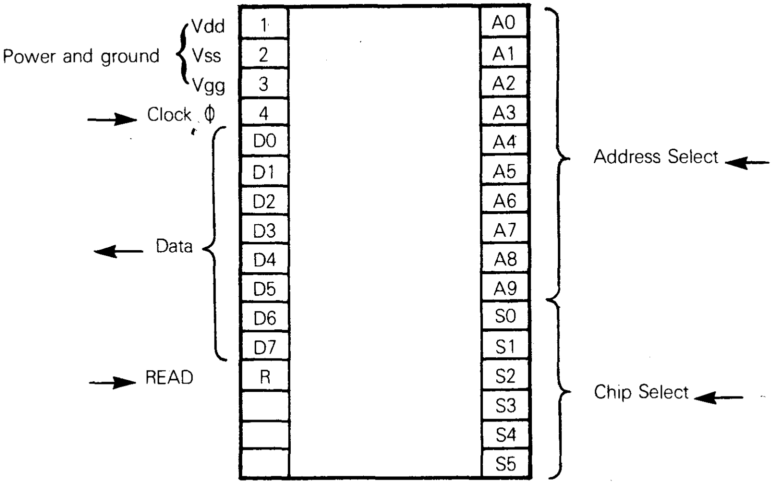


Figure 5-1. Read-Only Memory Chip Pins and Signals

ROM DEVICE SELECT

Notice that the 16 address signals of the CPU become ten word address signals and six device select signals at the ROM. This distribution of the address lines implies that the ROM has 1024 (2^{10}) words of memory, and a 6-bit select code. Providing the high order six address lines coincide with the 6-bit device code of the ROM, the ROM will decode the low order ten address lines as representing one of its 1024 memory words. Then, when the READ control signal goes true, ROM logic will place the contents of the addressed memory word at the data pins D0 — D7. If the six high order address lines do not coincide with the ROM select code, then the ROM will ignore current events.

Frequently a ROM chip will have no chip select logic; then there will be a single chip select signal, which must be generated by external logic. It is also very common for a ROM device to have two select inputs. For the ROM to be selected, one input must be low while the other input is simultaneously high.

If the ROM chip has its own select logic, then the select code is a permanent feature of the ROM chip design. If, for example, a ROM chip has the select code 001100, then this select code will have been burned into the ROM chip when it was created, with the result that the ROM chip will respond only to memory addresses 3000_{16} through $33FF_{16}$:

```
 Select
  Code      Lowest memory address
0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
   3       0       0       0

 Select
  Code      Highest memory address
0 0 1 1 0 0 1 1 1 1 1 1 1 1 1 1
   3       3       F       F
```

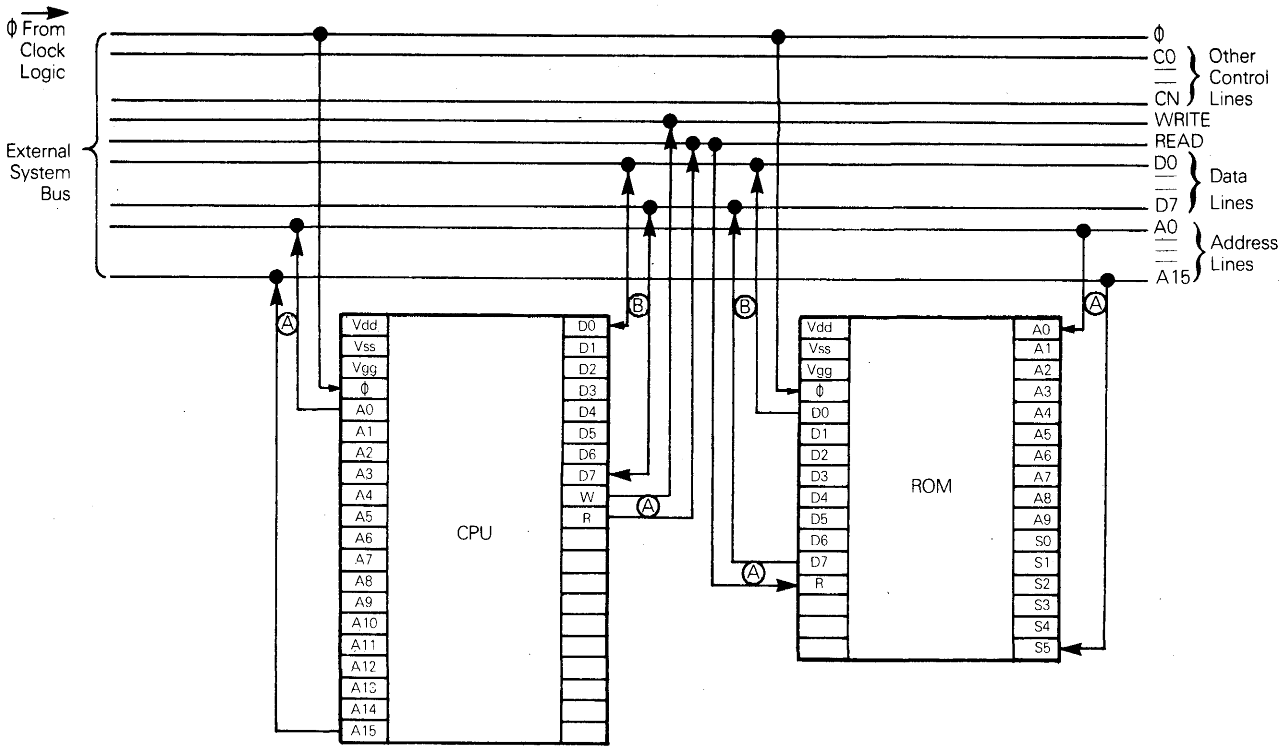
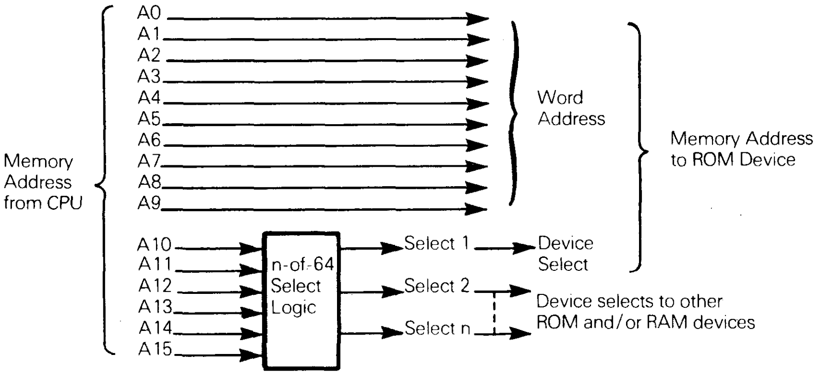


Figure 5-2. ROM And CPU Chips Connected Via External Data Bus

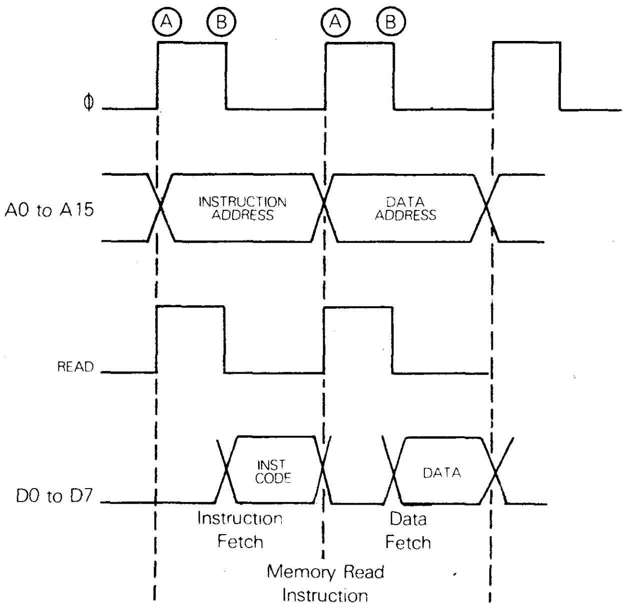
In other words, a ROM chip with one device select code must be looked upon as differing from an identical ROM with a different device select code.

If device select logic is external to the ROM chip, then the device select code is not a permanent feature of the ROM; merely modifying the external device select logic will change the range of addresses within which the ROM device responds to memory accesses. External device select logic may be illustrated as follows:



Logic that is internal to a ROM chip will never be of any concern to you, as a microcomputer user. The way in which the ROM chip selects or deselects itself, the way in which it responds to read control signals, the way in which it extracts necessary data and places the data at pins D0 - D7, are all completely irrelevant since there is nothing you can do about it.

Consider a memory read instruction; timing for this instruction is reproduced here, as it appeared in Chapter 4, but with the keying symbols (A) and (B) to link it to Figure 5-2.



Recall that so far as logic external to the CPU is concerned, **there is no difference between an instruction fetch operation or a data fetch operation.**

Each operation begins with clock Φ rising (at Ⓐ); at this time the CPU outputs an address on the address lines, and at the same time sets READ high. The ROM receives these signals via the External System Bus. If the high order six address lines (A10 - A15) coincide with the ROM select code (S0 - S5), then the ROM chip logic fetches the contents of the memory word addressed by A0 - A9, and places this data at D0 - D7.

By the time Φ goes low and READ goes low (at Ⓑ), ROM logic must have placed the requested data on D0 - D7; data must stay on these lines until Φ goes high again.

READ-WRITE MEMORY (RAM)

RAM interface logic is more complex than ROM interface logic. RAM logic must be able to take data off the data lines of the External System Bus and place this data in an addressed memory word; in addition, RAM logic must be able to extract data from an addressed memory word and place this data on the External System Bus data lines. Also, most RAM is implemented using a number of RAM chips, with each chip supplying one or more bits of the data word.

Using a number of chips to support a single data word is a simple enough concept; it means that each chip will only have one data pin, and this pin will be connected to one of the eight data lines, D0 - D7.

As described for ROM, RAM interface logic will partition the address lines A0 - A15 into a device select code and a memory address. However, there may be eight RAM chips (for an 8-bit word), each of which has the same device select code, but is connected to a different data line on the External System Bus. Figure 5-3 illustrates a single RAM chip with on-chip device select logic, and Figure 5-4 shows one way in which RAM memory may be added to the ROM-CPU combination illustrated in Figure 5-2.

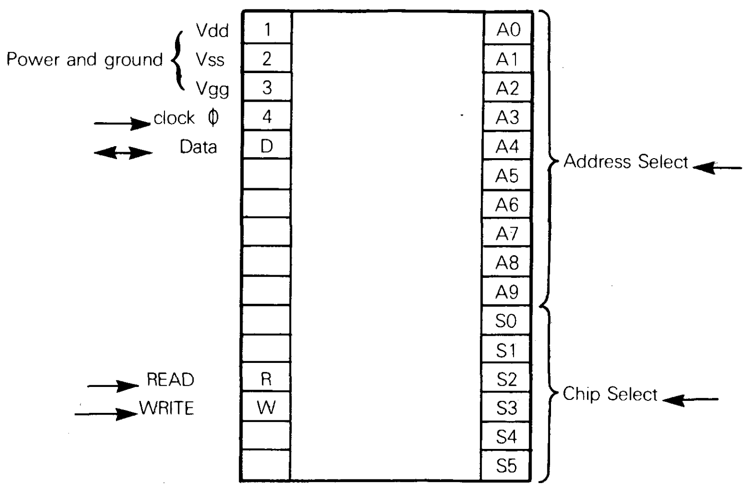


Figure 5-3. Read-Write Memory Chip Pins And Signals

Some microcomputers have special RAM interface logic devices. These devices may refresh dynamic RAM and/or provide device select logic. The Fairchild F8 needs special RAM interface devices because of its unique logic distribution. Figure 5-5 illustrates RAM controlled by a RAM interface device.

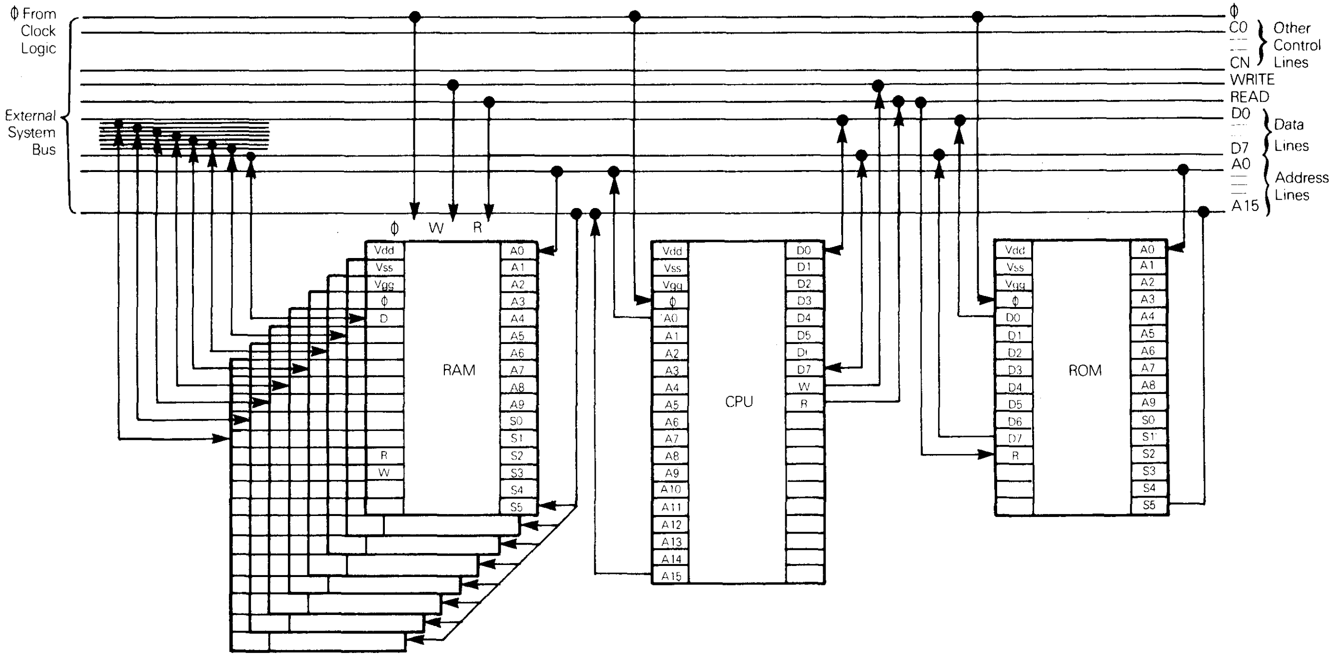


Figure 5-4. RAM (Without RAM Interface), ROM And CPU Chips Connected Via External Data Bus

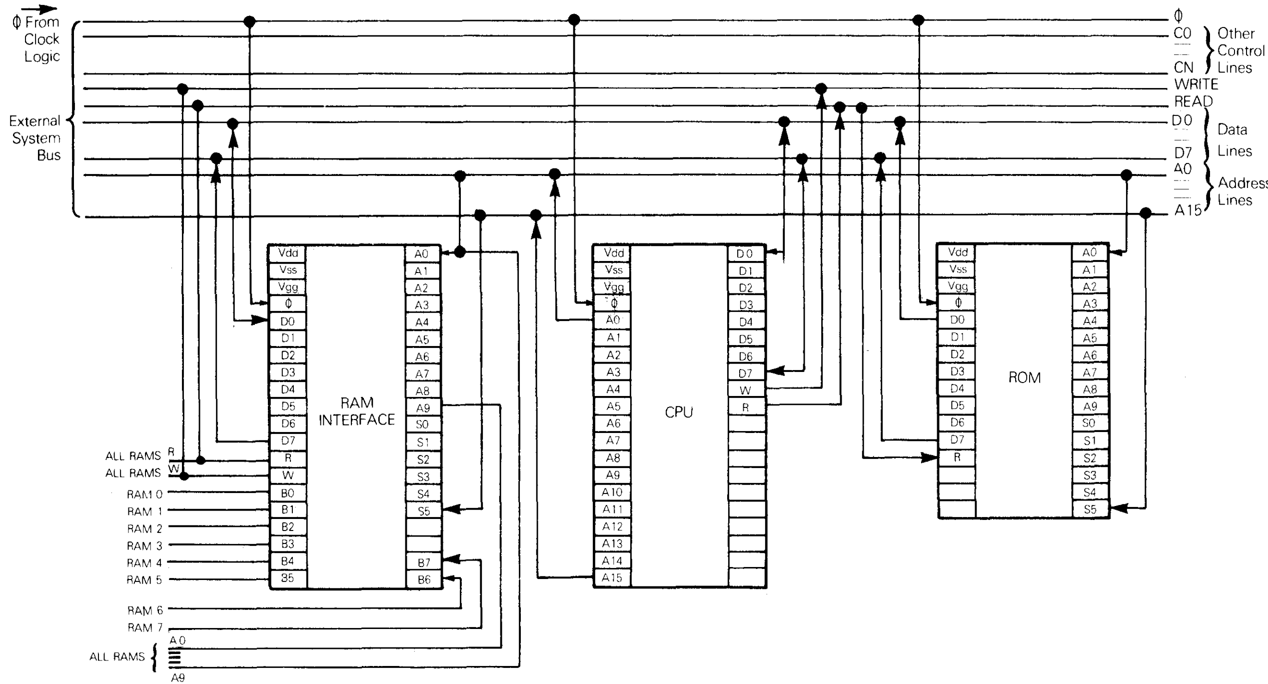


Figure 5-5. RAM Interface, ROM And CPU Chips Connected Via External Data Bus

TRANSFERRING DATA BEYOND THE MICROCOMPUTER SYSTEM (INPUT/OUTPUT)

The transfer of data between logic that is part of the microcomputer system and logic that is beyond the microcomputer system is generally referred to as Input/Output (I/O).

MICRO-COMPUTER SYSTEM BOUNDS

We will include, within the boundary of the microcomputer system, all logic that has been *specifically designed to operate* in conjunction with the CPU. We will classify all other logic as external.

The interface between the microcomputer system and external logic must be clearly defined; it must contain provisions for transfer of data, plus control signals that identify events as they occur.

There are many ways in which data transfer between the microcomputer system and external logic can be accomplished; but they all fall into the following three categories:

1) **PROGRAMMED I/O.** In this case, all data transfers between the microcomputer system and external logic are completely controlled by the microcomputer or, more precisely, by a program being executed by the microcomputer CPU.

   There will be some well defined protocol whereby the microcomputer system gives evidence that data being output has been placed in a location where external logic can access it; or, alternatively, the microcomputer system will indicate in some predefined way that it is waiting for external logic to place data in some predefined location from which it can be input to the microcomputer system.

   The key characteristic of programmed I/O is that external logic does as it is told.

2) **INTERRUPT I/O.** Interrupts are a means for external logic to force the microcomputer system to suspend whatever it is currently doing in order to attend to the needs of the external logic.

3) **DIRECT MEMORY ACCESS.** This is a form of data transfer which allows data to move between microcomputer memory and external devices without involving the CPU in data transfer logic.

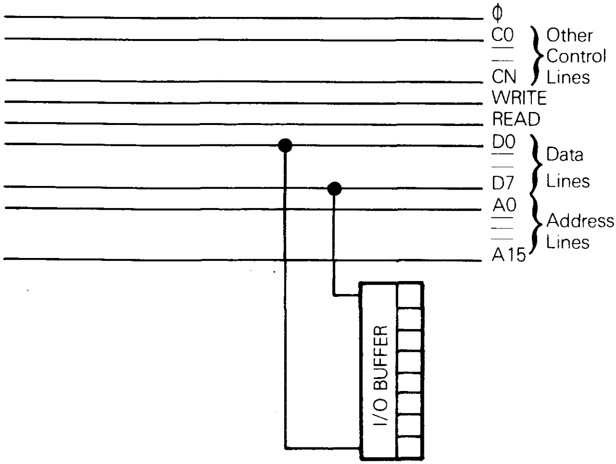
The physical requirements for each type of I/O will be described in turn.

PROGRAMMED I/O

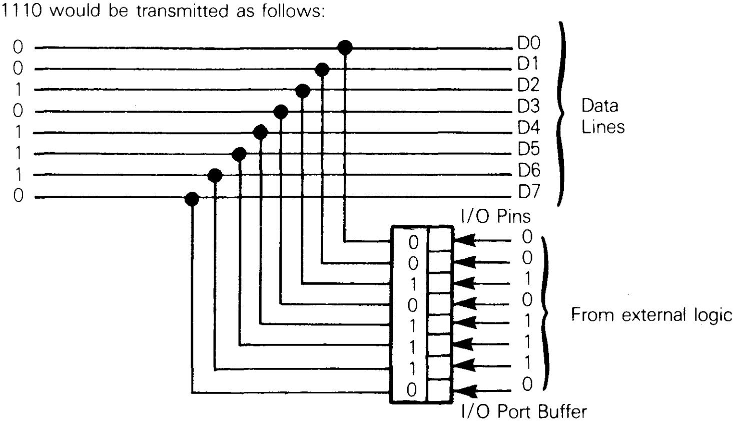
Data are transferred between a microcomputer system and external logic, in either direction, via an I/O port.

I/O PORTS

An I/O port will consist of an I/O Port Buffer, connected to the data lines of the External System Bus and to pins which access external logic:



When external logic transmits data to the microcomputer system, it does so by presenting the data at the I/O port pins, whence the data are stored in the I/O buffer. The binary value 001001110 would be transmitted as follows:



The I/O Port Buffer cannot be constantly communicating with the data lines of the External System Bus, as illustrated above, since the data lines of the External System Bus may be carrying data to or from memory. If the I/O port were permanently communicating with the data lines of the External System Bus, then every time external logic presented data at the I/O pins, this data would be propagated down the shared data lines with unpredictable consequences.

The microcomputer CPU will therefore select an I/O port and read the contents of the I/O Port Buffer, in much the same way as data gets read out of memory. This parallel between reading data out of I/O Port Buffers and reading data out of memory is appropriate, since most microcomputer systems transfer a great deal of data to and from external logic; therefore, they have more than one I/O port.

I/O PORTS ADDRESSED USING MEMORY ADDRESS LINES

We can develop a parallel I/O device with one or more I/O ports, where the I/O Port Buffers have addresses, just as memory words have addresses. A simple scheme would be to take the high order address line (A15) and design microcomputer logic such that whenever this line is 0, a memory module is selected, but whenever this line is 1, an I/O Port Buffer is selected. In other words, memory addresses of $7FFF_{16}$ and below will access memory words, whereas memory addresses of 8000_{16} and above will access I/O Port Buffers. Using the READ and WRITE control lines, Figure 5-6 illustrates a parallel I/O device with one port.

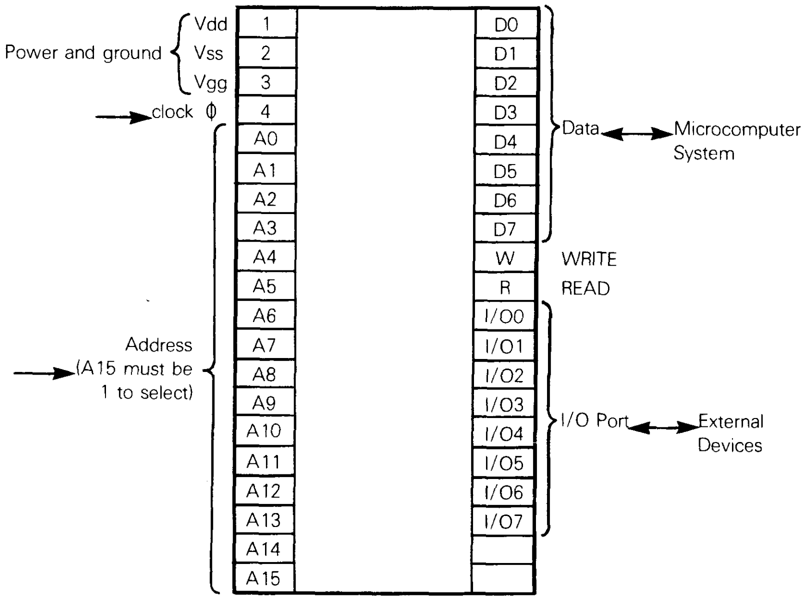


Figure 5-6. A Single Port, Parallel I/O Interface Device

The device in Figure 5-6 is referred to as a parallel I/O device because data is written and read in eight, simultaneous, parallel binary units.

Note that there is no reason why an I/O device should have only one I/O port. The number of I/O ports that a parallel I/O device has is purely a function of the number of pins that are economically available on a dual in-line package. The device illustrated in Figure 5-6 uses its pins as follows:

1) Sixteen pins are connected to the address lines of the External System Bus and provide the information needed to determine if this I/O Port Buffer has been selected.

2) Eight pins are connected to the data lines of the External System Bus and are used to transfer information from the External System Bus to the I/O Port Buffer, or from the I/O Port Buffer to the External System Bus.

3) Two control lines, READ and WRITE, determine whether data will flow from the I/O Port Buffer to the External System Bus (read), or from the External System Bus to the I/O Port Buffer (write).

4) Three pins are required for power and ground.

5) One pin is required for the clock signal.

That sums to 30 pins, which only leaves ten pins available for I/O ports. Therefore, the parallel I/O device illustrated in Figure 5-6 can only support one I/O port.

There is, of course, an obvious way to increase the number of I/O ports on our parallel I/O device. Having 16 address lines implies that the microcomputer system is going to address 32768 (2^{15}) I/O ports and that is unlikely.

How about reducing the number of address lines to ten? Now 16 pins are available for I/O ports and our parallel I/O device can have two I/O ports, as illustrated in Figure 5-7.

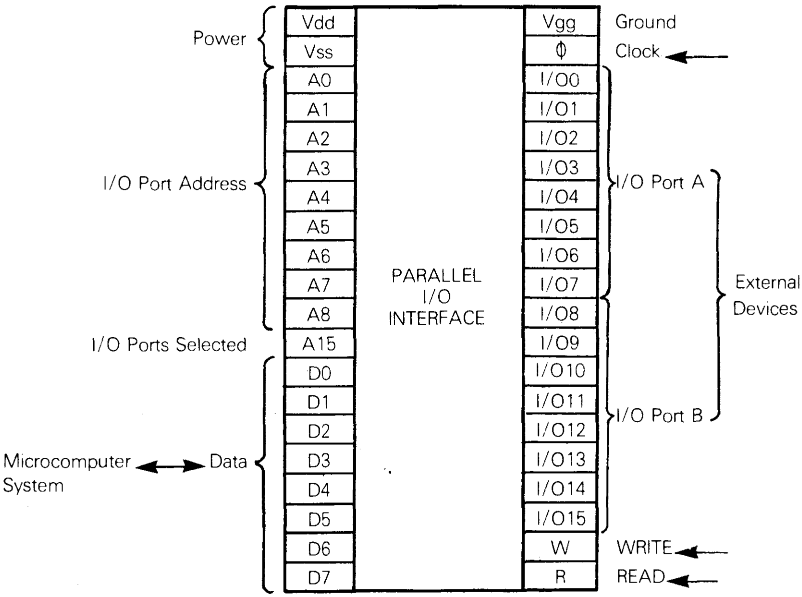
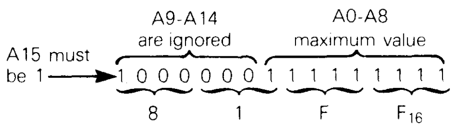


Figure 5-7. A Two-Port, Parallel I/O Interface Chip

The ten address lines in Figure 5-7 will be divided as follows:

1) One of the ten address lines will be A15, since this line must be 1 in order to select an I/O port rather than a memory word.

2) The remaining nine address lines may be any nine of the other 15 address lines. For example, if Address lines A0 — A8 are connected, then addresses 8000_{16} through $81FF_{16}$ will select I/O ports:



In the unlikely event that this arrangement provides insufficient I/O port addresses, address lines A9-A14 could be included as part of the address transmitted to another parallel I/O device.

The penalty paid, when the high order address line (A15) is used to select I/O Port Buffers, is that only 32768 (2^{15}), rather than 65536 (2^{16}) words of memory can be addressed. In the world of minicomputers, this reduction of addressable memory can be a severe price to pay, just to simplify I/O Port Buffer selection. **In the world of microcomputers, the penalty is not very significant, since very few microcomputers require 32768 words of addressable memory (or anywhere close to that much memory).** Typically, a microcomputer application will require a total of between 1024 and 4096 words of memory.

I/O PORT ADDRESSES

Nevertheless, many microcomputer systems use separate addressing logic to select I/O ports. Figure 5-8 illustrates one possible scheme which adds two control lines to the microcomputer CPU. One control line, IOSEL, specifies that address lines A0 — A7 contain the I/O Port Buffer select code. The other control line, IORW, if high, indicates that the External Data Bus lines contain information which must be read into the I/O Port Buffer. If IORW is low, then the selected I/O Port Buffer contents must be output to the data lines of the External Data Bus.

As you will discover in Chapter 7, the two methods we have described for selecting I/O ports are just two out of a bewildering array of possibilities. These two methods do, however, broadly cover the most common ways in which I/O ports are addressed.

Unfortunately, the blind transfer of data between a microcomputer system and external logic will not always provide sufficient I/O capability. The following necessities are missing:

1) **The microcomputer system must be able to tell external logic when data has been placed in an I/O buffer and is therefore ready to be sampled. Conversely, external logic must have means of indicating to the microcomputer system that it has placed data in an I/O buffer and the data can now be read.**

2) **The microcomputer system, and external logic, must each have some way of informing the other as to the nature of the data placed in an I/O buffer.** Clearly data being transferred between the microcomputer system and external logic is subject to various interpretations. For example, it may be pure numeric data; but on the other hand it may be a code identifying operations to be performed, or already accomplished. It may also be part, or all of an address.

I/O CONTROL

When the microcomputer outputs signals to external logic as a means of identifying events or data, these signals are referred to as I/O controls. The same information travelling in the opposite direction, that is, from external logic to the microcomputer, is referred to as I/O status. The differentiation of information into controls and status, based upon the direction of the information, is logical, since we are dealing with a situation where the microcomputer is at all times in control of events. In other words, the microcomputer outputs controls to control external logic sequences. External logic cannot control the microcomputer; it can only input status information for the microcomputer to interpret when and how it sees fit.

I/O STATUS

Minicomputer systems will usually have a whole set of I/O control and status lines that are separate and distinct from I/O ports. Microcomputers more commonly allocate one or more I/O ports to function as control and status conduits, while separate I/O ports transfer data.

I/O Port A in Figure 5-8 might be used to transfer data, while I/O Port B is used to transfer control and status information. So far as the microcomputer system is concerned, the same instruction sequences are used to handle data flow through either I/O port. It is the way the microcomputer system interprets a data word that determines whether the word is data, control or status information.

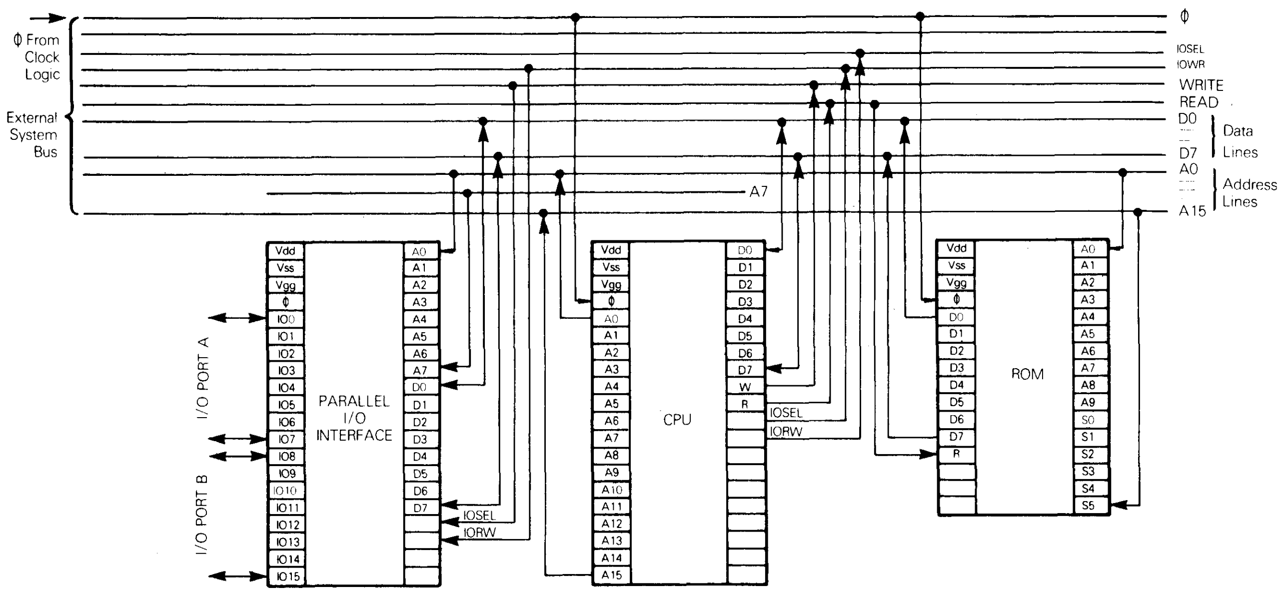


Figure 5-8. Parallel I/O Interface Chip
Using I/O Addressing Logic

INTERRUPT I/O

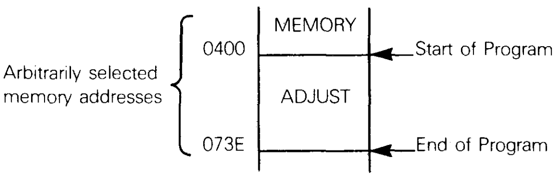
Most microcomputer CPUs have a control signal via which external logic can demand the attention of the microcomputer system. This signal is referred to as an interrupt request signal because, in effect, the external logic is asking the microcomputer system to interrupt whatever it is currently doing in order to service more pressing external logical needs.

THE CONCEPT OF AN INTERRUPT

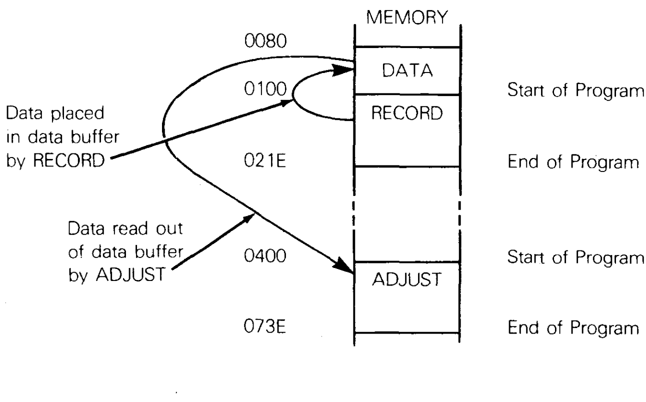
We will begin the discussion of interrupts with an example that is too simple to be realistic, but contains all the key features of a meaningful application.

Suppose a microcomputer system is being used to control the temperature of shower water, as illustrated in Figure 5-9. A thermometer measures the temperature of the mixed hot and cold water issuing from the shower head and transmits this temperature, as a digital signal, to the microcomputer system. The microcomputer system compares this temperature to a set point, which is supplied by an appropriate control. Depending on the difference between the real and desired shower temperature, the microcomputer system outputs data which must be interpreted as a valve control signal, causing the valve to increase or decrease the hot water flow.

There are a number of reasons why this simple sounding application is, in reality, far from simple. As experience will have taught you, there is some delay between the time you adjust a shower tap and the time that the water issuing from the shower head changes temperature. For this reason, **a non-trivial program will have to be executed by the microcomputer to ensure that it does not attempt to make ridiculous adjustments. We will call this program ADJUST, and illustrate it residing in program memory as follows:**



Another program, called RECORD, will input data from the temperature sensor, correctly interpreting the data to represent temperature readings. The only contact between programs RECORD and ADJUST are that ADJUST will anticipate finding data in a certain area of data memory and RECORD will place the data in correct format, in that required area. Our memory now looks like this:



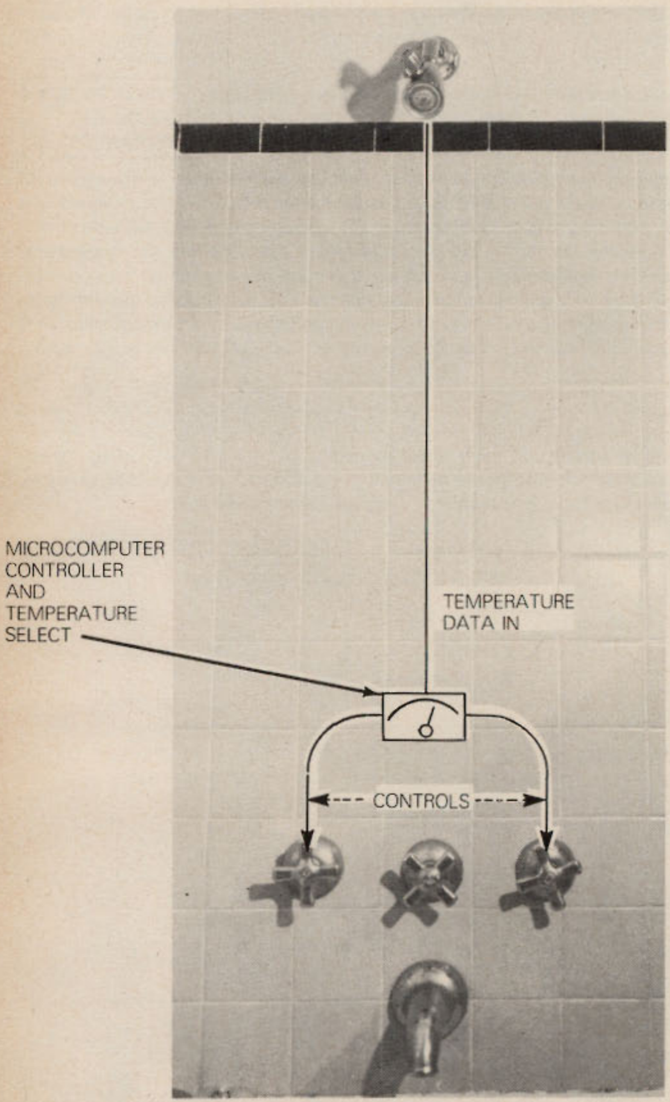


Figure 5-9. A Microcomputer Controlling The Temperature of Shower Water

The way in which shower head temperatures are read and transmitted to the microcomputer system is another feature of this problem which is not as straightforward as might appear. It will take approximately half a second for an inexpensive temperature sensor to record a temperature. Half a second may not seem like very much time, but a microcomputer can execute approximately a quarter of a million instructions during this time period.

HANDLING AN INTERRUPT REQUEST

How is the microcomputer going to know when the temperature sensor has a new value to transmit? If the temperature sensor simply tries to send data to an I/O port, the microcomputer system is very likely not to read the temperature. One reading can easily get lost in a quarter of a million instruction executions.

One way of resolving the problem is illustrated in Figure 5-10. **A three-step sequence allows the temperature sensor to call the microcomputer system's attention as follows:**

INTERRUPT REQUEST

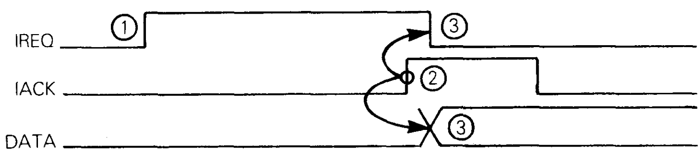
① **The temperature sensor transmits an interrupt request signal (IREQ) to the microcomputer system via an External System Bus control line.**

② **The microcomputer system has the choice of accepting or rejecting the interrupt request; it accepts the interrupt request by outputting an interrupt acknowledge signal (IACK) on an External System Bus control line.**

INTERRUPT ACKNOWLEDGE

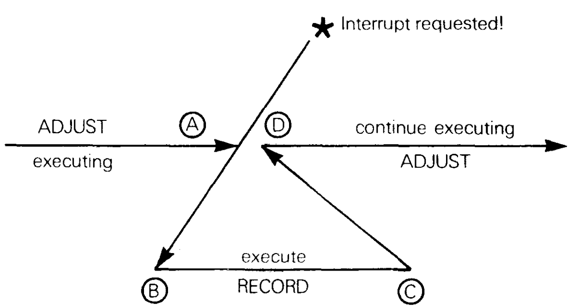
③ **The external device uses the interrupt acknowdge signal as an enable signal, to transmit data to I/O Port A. Also, the external device must remove its interrupt request signal upon receiving an interrupt acknowledge since, clearly once the interrupt request has been serviced, the external device is no longer requesting another interrupt.**

Timing for this three-step sequence may be illustrated as follows:



Note that although we have been talking about the external device inputting data to the microcomputer system, data flow could just as easily be in the opposite direction; in fact, there is no reason why any data flow need follow an interrupt. The program executed following an interrupt could, for example, simply output control signals.

The purpose of the interrupt is to tell the microcomputer that it must suspend whatever it is doing, process the data being input, then carry on with its suspended operations. With reference to programs RECORD and ADJUST, this is what happens:



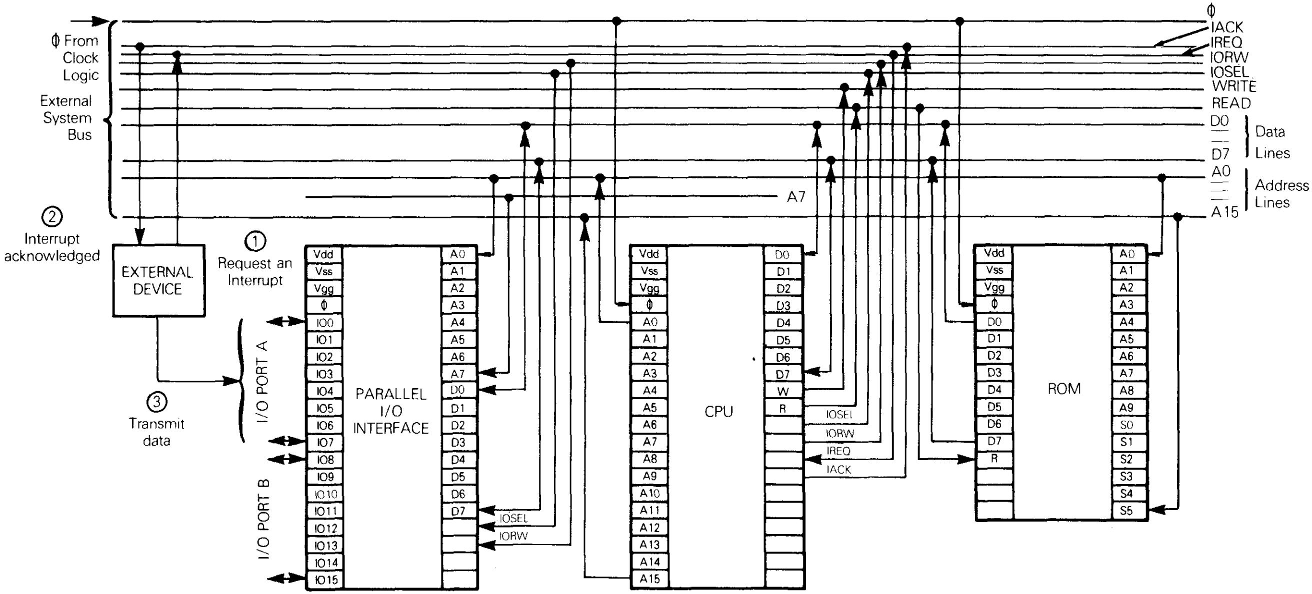


Figure 5-10. An External Device Using An Interrupt Request To Let The Microprocessor Know That Data Is Ready To Be Input

Refer again to Figure 5-10. Steps (1) and (2) cause the interrupt request to be sensed by the microcomputer at (A) above. The microcomputer CPU responds by suspending execution of ADJUST, while executing RECORD ((B) to (C)). While RECORD is executing, the data transmitted by the temperature sensor ((3) in Figure 5-10) is read into the microcomputer system, since program RECORD has been written to anticipate arrival of this data.

When RECORD has completed execution at (C), execution of ADJUST continues at (D), picking up exactly where it left off at (A).

The key feature of program RECORD's execution is that it is an unscheduled event. There is no logic within the microcomputer system that can predict when or how often program RECORD will be executed. However, there is logic within the microcomputer system that can suspend any program's execution, later restarting execution from the exact point of suspension.

A MICROCOMPUTER'S RESPONSE TO AN INTERRUPT

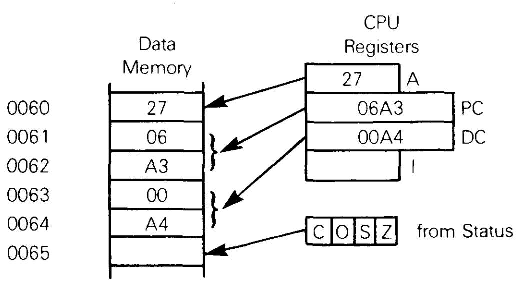
At its most elementary level, a microcomputer CPU could respond to an interrupt request by simply loading into the Program Counter the starting address of the program that external, interrupting logic wants executed. But that begs two questions:

1) **What happens to the program that was being executed?**
2) **Where does the microcomputer CPU get the address of the program which the interrupting logic wants executed?**

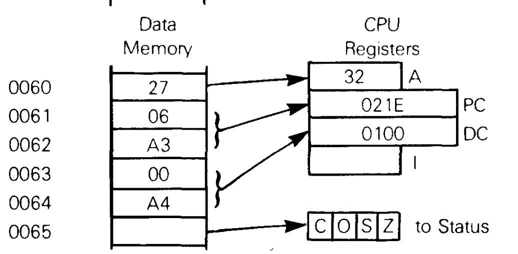
Consider first what happens to the program that was being executed.

SAVING REGISTERS AND STATUS

The old program may have important information in the status flags, the Data Counter and Accumulator; this data is going to be wiped out by the new program, so that when the new program has finished executing, the old program will no longer be able to restart. This problem is resolved by saving the contents of all CPU registers, including the Program Counter, before starting to execute the new program. When the new program completes its execution, the saved Program Counter value is the address of the instruction that was about to be executed when the old program was interrupted; so, by merely restoring the saved values into the CPU registers, the old program can pick up where it left off. This concept is illustrated as follows:



This is the situation when ADJUST is interrupted to execute RECORD (registers contents and memory addresses have been arbitrarily selected)



This is the situation when RECORD has finished executing and ADJUST must continue where it left off. (Registers contents are again arbitrarily selected.)

An interrupt will not be acknowledged until the current instruction has completed executing. This being the case, there is no need to save the contents of the Instruction register, since it contains an instruction code which has been processed. In other words, the interrupt directly precedes the arrival of a new instruction code.

There are two ways in which the old program status flags and registers' contents may be saved, as illustrated above, before the new program starts execution. One way would be for the interrupt request signal to initiate execution of a microprogram (stored in the Control Unit), which simply writes the contents of CPU registers into a data area of memory which has been set aside for this purpose. A microcomputer designer may be reluctant to use up precious Control Unit space in this way, and instead may require the programmer to write a short program which will do the same thing. Such a program is called an "Interrupt Service Routine."

INTERRUPT SERVICE ROUTINE

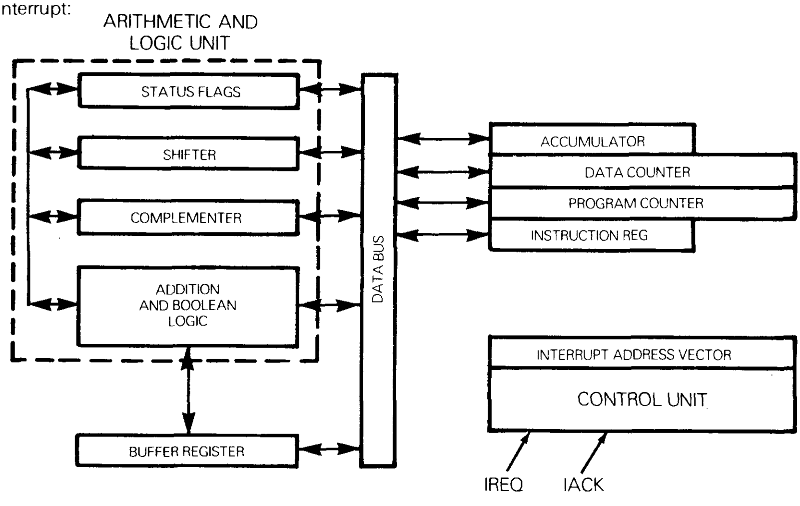
At the end of the new program's execution, whatever logic was used to save the old program's registers' contents must be executed in reverse, to restore the old program's registers and status.

We will consider Interrupt Service Routines in more detail in Chapter 6, after we have discussed programming in more detail.

INTERRUPT ADDRESS VECTOR

Now consider how the microcomputer CPU gets the address of the program which the interrupting logic wants executed. We will refer to this as the Interrupt Address Vector.

There are almost as many ways of determining which program must be executed following an interrupt request as there are microcomputers. In the case of our shower temperature control problem, there is an easy solution. The shower head temperature sensor is the only external device that can request an interrupt, and there is only one program it can want executed following the interrupt. This being the case, the microcomputer CPU could be built with internal logic that causes one program, origined at one specific memory address, to be executed following an interrupt:



Now every time the Control Unit receives an interrupt request (IREQ) and it is ready to service the interrupt, it does as follows:

1) **Send out the interrupt acknowledge signal, IACK.**

2) **Save the contents of Status Flags, the Accumulator, the Data Counter and the Program Counter, or else allow the programmer some way of doing the same thing.**

3) Move the contents of the Interrupt Address Vector to the Program Counter.

A minicomputer programmer would consider this method of responding to interrupts as laughably ridiculous. Who knows when and how the minicomputer may next have to respond to an interrupt? To specify that all interrupts will be serviced by a program origined at memory address 0400_{16} (for example) would be an intolerable restriction, because it reduces the minicomputer programmer's ability to be flexible.

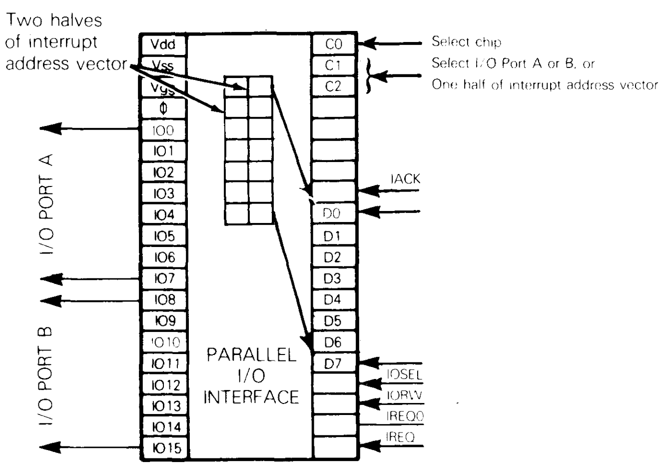
In microcomputer applications, having fixed interrupt address vectors makes a certain amount of sense. Remember that microcomputers are going to be used as logic components, not as general purpose computers. Most microcomputers are used in dedicated, non-varying situations, where one, or a few specific interrupts will require equally specific responses from the microcomputer system.

Figure 5-11 shows how the parallel I/O interface device and the ROM device from Figure 5-8 could be modified to receive the interrupt request signal IREQ. Each device, as modified in Figure 5-11, contains a 16-bit register, in which an interrupt address vector is permanently stored. Upon receiving the interrupt acknowledge signal IACK, the device transmits an interrupt address vector to the CPU, via the address lines of the External System Bus. After saving the contents of status flags and CPU registers, the CPU loads the interrupt address vector into the Program Counter.

The Parallel I/O Interface device, only having eight address pins attached to A0 - A7, will have to transmit its interrupt address vector to the CPU in two halves; and the CPU Control Unit microprogram will have to load each half appropriately into the Program Counter. This is referred to as multiplexing lines, that is, using the same bus lines to carry signals that must be interpreted in different ways at different times.

MULTIPLEXED LINES

Notice that we have a problem with the Parallel I/O interface device, as illustrated — we have run out of pins. IACK has to be input somehow in response to IREQ0. The best way of resolving this problem is to replace the address pins A0 - A7 with three chip select pins CO, CI and C2. That leaves five unused pins, one of which can be assigned to IACK:



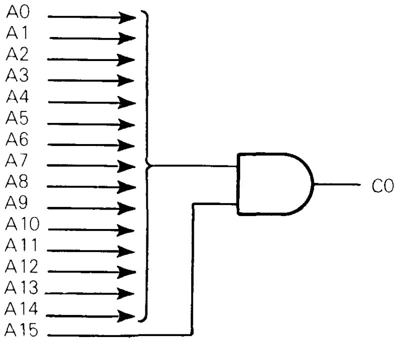
The Parallel I/O interface device changes in two important ways.

First, CO, CI and C2 will be the product of additional chip select logic, which receives some or all of the external system bus address lines as inputs.

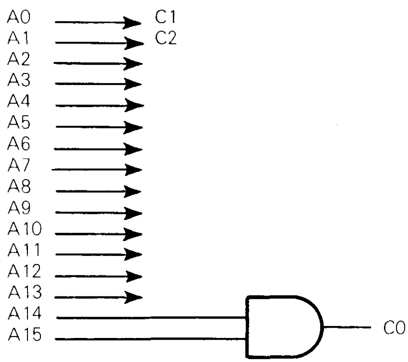
Second, the data bus is now multiplexed; at different times it may transmit data or an address to the CPU.

SELECTING I/O DEVICES

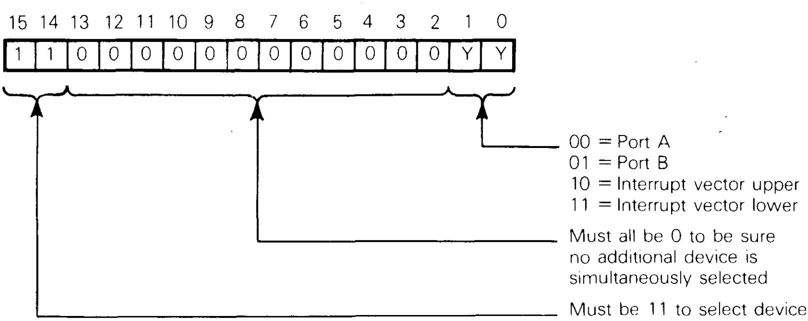
Having external chip select logic is the rule rather than the exception among real microcomputer support devices. In reality, the external select logic frequently does not exist. Since there are very few I/O devices in a typical microcomputer system, and more than 32K bytes of memory is rare, you could create C0 as the AND of A15 and any other address line:



Any two of the 14 unused address lines can be tied to Cl and C2, in order to select one of the four addressable locations on the Parallel I/O interface device. Consider this example:



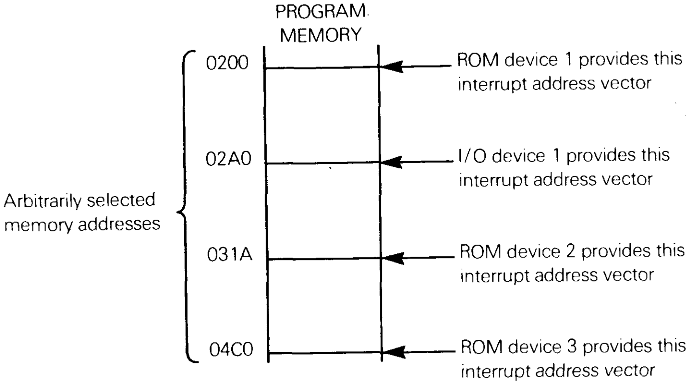
Memory addresses may select ports as follows:



Thus $C000_{16}$ addresses Port A
$C001_{16}$ addresses Port B
$C002_{16}$ addresses Interrupt Vector Upper
$C003_{16}$ addresses Interrupt Vector Lower

Multiplexing the data bus is not a real problem; it **simply means the interrupt service routine** — the program executed immediately following the interrupt acknowledge — **must read the contents of memory locations $C002_{16}$ and $C003_{16}$,** and treat these two data bytes as the starting address for the program to be executed next.

Having the interrupt request signal arrive at memory or interface devices, rather than at the CPU, means that for every memory or interface device in the system, a different external device can have its own interrupt service routine identified for post-interrupt execution. This concept is illustrated as follows:



In the above illustration, interrupt request signals arriving at ROM devices 1, 2 or 3 will always specify execution of programs stored in memory with execution addresses 0200_{16}, $031A_{16}$ and $04C0_{16}$, respectively. An interrupt request signal arriving at Parallel I/O Interface device 1 will always specify execution of a program stored in memory with execution address $02A0_{16}$. Each interrupt request arrives as a separate and distinct IREQ signal; it is passed on to the CPU as IREQ0. When the CPU acknowledges with IACK, the ROM or I/O device requesting the interrupt transmits the interrupt address vector, which is a permanent feature of the ROM or I/O device.

But a new problem arises when more than one device capable of requesting an interrupt is included in a microcomputer system. What happens when more than one device simultaneously requests an interrupt? Which device is to be acknowledged, and how do we prevent the other devices from also acknowledging?

There are two parts to the answer. First, we must provide devices with a means of identifying themselves; we use select codes for this purpose. Next we must include interrupt priority arbitration logic.

Let us examine these two concepts.

INTERRUPTING DEVICE SELECT CODES

Consider an improved model of our shower temperature controller, designed to control the temperatures of many showers for motels and hotels. A microcomputer would certainly be fast enough to monitor and control shower head temperatures for 100 or more showers.

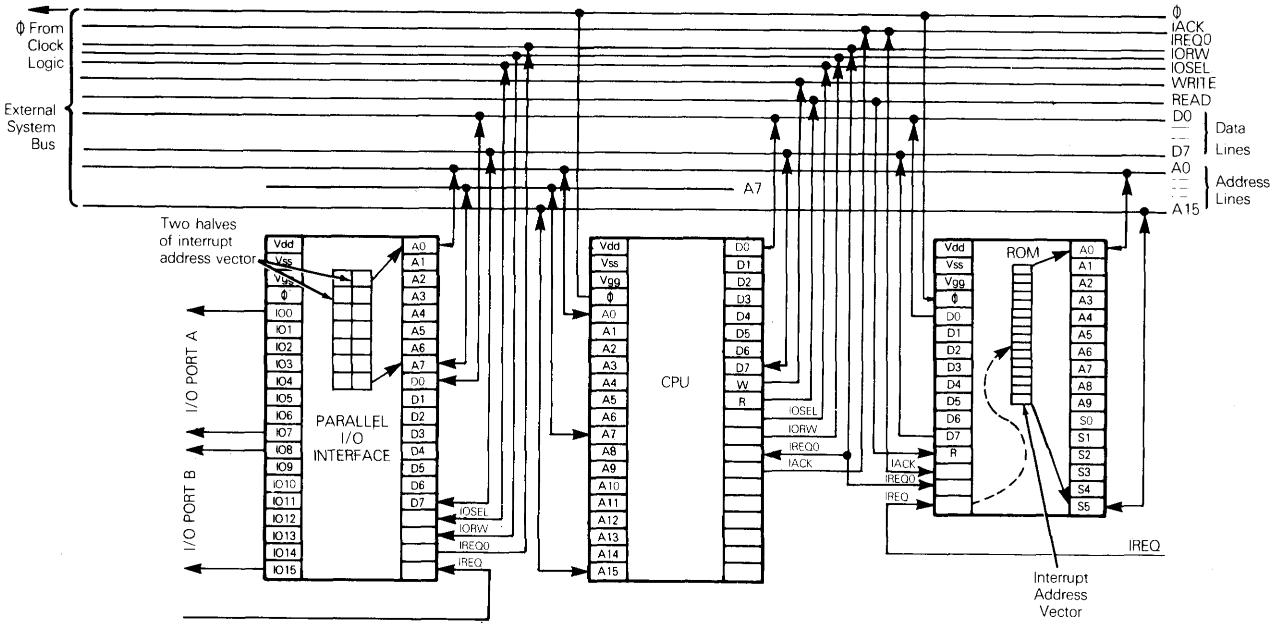


Figure 5-11. Using I/O And ROM Chips To Handle Interrupts

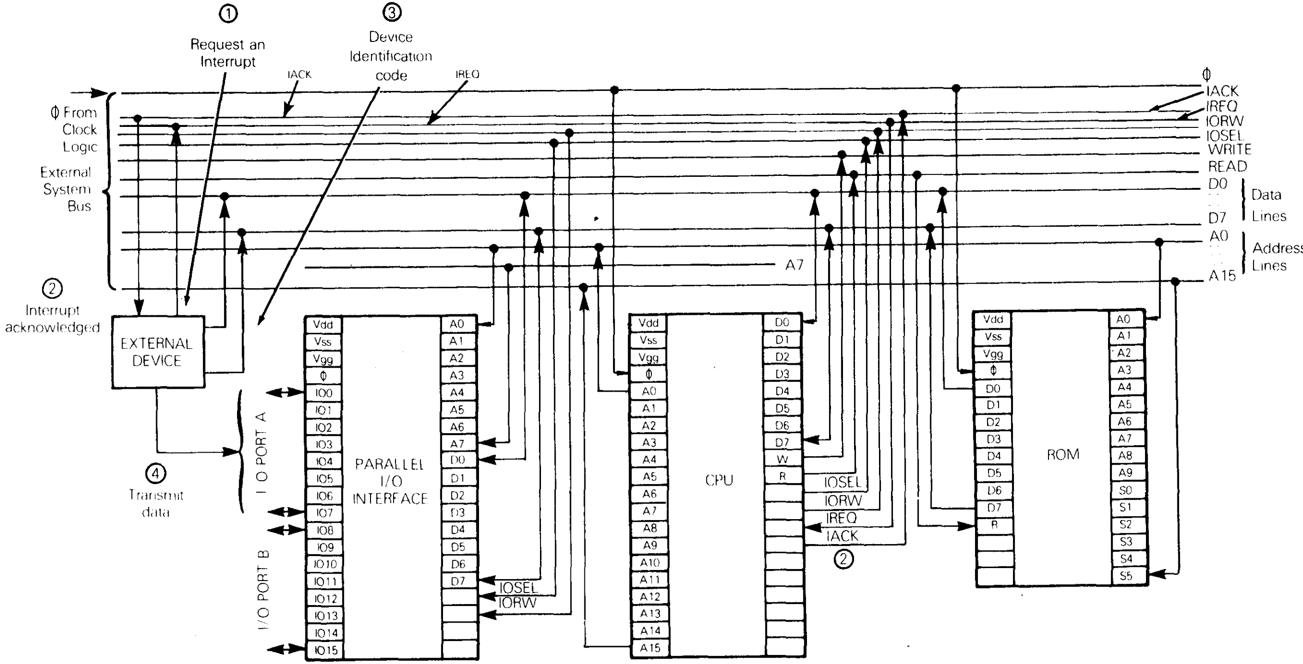


Figure 5-12. An External Device Using An Interrupt Request And A Device Identification Code To Let The Microcomputer Know That Data Is Ready To Be Input.

It would not be very economical or practical to require a new memory or I/O interface device to be added to the microcomputer system for every new shower to be controlled. Many microcomputers will therefore require an external device, when requesting an interrupt, to accompany the request with an identification code. Figure 5-12 shows how an external device may connect directly to the External System Bus, placing its identification code on the data lines of the External System Bus when the CPU acknowledges its interrupt request with IACK. With reference to Figure 5-12, events proceed as follows:

① External device logic creates an interrupt request signal, which it transmits to the CPU as IREQ.

② When the CPU is ready to service the interrupt request, it responds by outputting IACK.

③ Upon receiving IACK, external-device logic places an 8-bit select code on the data lines of the External System Bus. The CPU receives this data and interprets it as an external device identification code.

④ Following protocol specified by the microcomputer system, the external device places its data at a Parallel I/O Interface device's I/O port.

The idea of having external devices identify themselves with a code, as illustrated in Figure 5-12, makes a lot of sense to a minicomputer user (or designer), but to the microcomputer user it has one elementary flaw: it demands intelligence of the external device. Remember that a minicomputer may cost thousands of dollars and may be part of a system costing tens of thousands of dollars. Very few microcomputers cost more than $100, so we can only justify using a few dollars worth of logic to generate a device select code.

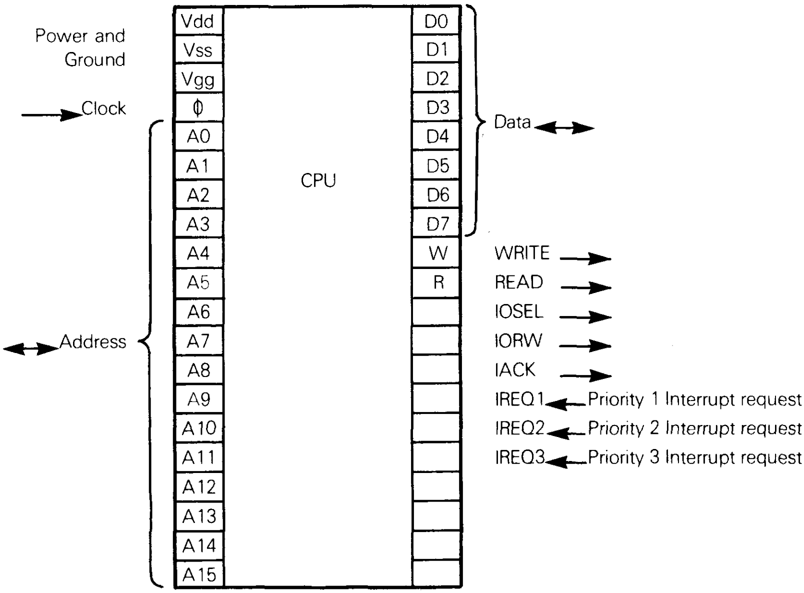
Here we have another reason why minicomputers and microcomputers are, and are likely to remain, fundamentally different. The cost of providing external devices with the logic implied by Figure 5-12 may only be a few dollars, which is trivial in the world of minicomputers. But a ROM device, or a Parallel I/O interface device, also only costs a few dollars; therefore, every demand for dedicated intelligence in external devices is, comparatively, an expensive demand in the world of microcomputers. External device select codes, which are so obvious in the world of minicomputers, are an expense which must be justified on a case-by-case basis in the world of microcomputers. Chip costs increase very little with chip complexity, so economics demand that the microcomputer system does as much as possible, and demands as little as possible of external logic.

Consider a simple example. Two minicomputers are priced at $3250 and $3640. Options that are not easily compared make the two prices hard to evaluate, in determining which minicomputer is really more expensive. If one minicomputer system requires that an external device have $10 worth of extra logic in order to request interrupts, this extra expense will only have a limited impact on overall economics.

Two microcomputer systems, configured for one very specific application, cost $53 and $61, respectively. If an external device will need $10 worth of extra logic to request an interrupt on the $53 microcomputer system, but not on the $61 microcomputer system, then the $53 microcomputer system may well be eliminated for this single reason.

INTERRUPT PRIORITIES

What happens when more than one external device requests an interrupt at the same time? This problem can be resolved in two ways. First, logic within the microcomputer system can have a number of interrupt request lines with ascending priorities, as follows:



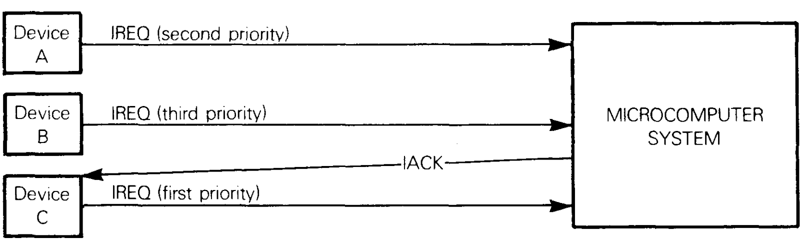
A special Interrupt Priority device could do the same job, using fewer CPU pins. Consider the Interrupt Priority device illustrated in Figure 5-13.

INTERRUPT PRIORITY CHIP

Before discussing how the Interrupt Priority device illustrated in Figure 5-13 works, we will define what is meant by interrupt priorities.

INTERRUPT PRIORITIES AND WHAT THEY MEAN

Suppose more than one external device may request an interrupt. If two or more external devices request interrupts by SIMULTANEOUSLY sending IREQ signals that overlap in time, then WHICH external device gets the interrupt acknowledge (IACK) is determined by interrupt priority:



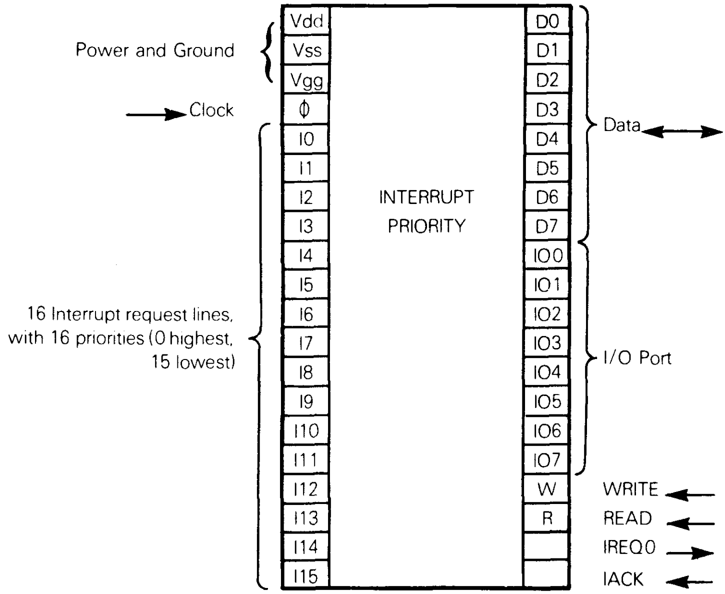
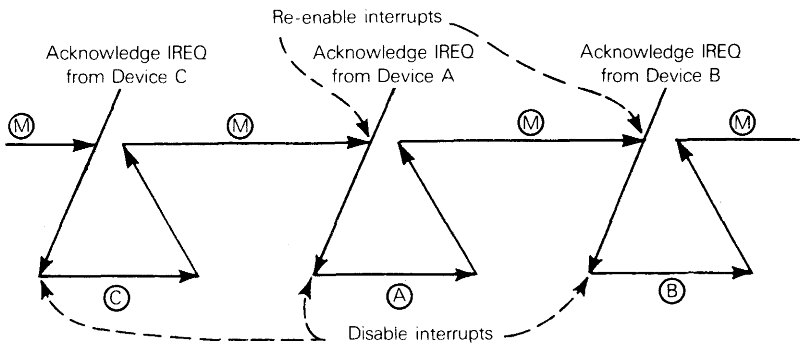


Figure 5-13. An Interrupt Priority Device

In the above illustration, external devices A, B and C all request interrupts by simultaneously transmitting IREQ signals to the microcomputer system. By whatever priority arbitration technique the microcomputer system is using, it is determined that Device C has the highest priority, Device A has the second priority and Device B has the lowest priority. The single acknowledge signal, IACK, must therefore be sent to Device C.

The fact that Devices A and B did not have their interrupt requests acknowledged does not imply that they must remove their IREQ interrupt request signals. They can do so if they wish. If they do not, they will be acknowledged, in turn, when the microcomputer CPU is subsequently ready to acknowledge interrupts again.

Refer again to the illustration of Devices A, B and C, all simultaneously requesting interrupt service. There are three interrupt service programs residing in memory, one for each device. These programs may be executed as follows:

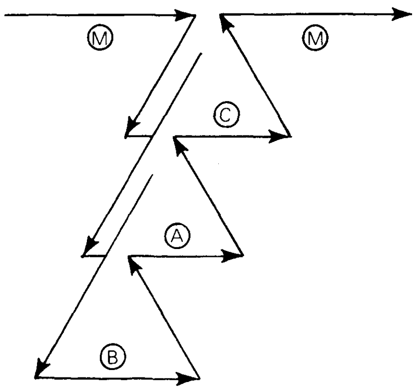


Ⓜ represents the main program which is executing, and which gets interrupted. First Device C's interrupt service routine gets executed at Ⓒ. Subsequently Device A and B's interrupt service routines get executed at Ⓐ and Ⓑ, respectively.

If interrupt service routines Ⓒ, Ⓐ and Ⓑ are to be executed sequentially, as illustrated above, then while Ⓒ is executing, the microcomputer system's interrupt handling logic must be disabled, so interrupt requests (IREQ) from Devices A and B are ignored. At some point, after Ⓒ completes execution and Ⓜ has resumed executing, the microcomputer system interrupt logic is re-enabled; now IREQ from Device A is acknowledged. While Ⓐ is executing, the microcomputer system's interrupt handling logic is again disabled until some time when Ⓜ has resumed execution. Since Device B is still requesting an interrupt, Ⓑ now gets executed.

Special instructions are used to enable and disable interrupt logic in microcomputer systems. These instructions, and how they should be used, are described in Chapter 7.

Suppose the microcomputer system did not disable its interrupt logic while executing interrupt service routines such as Ⓐ, Ⓑ and Ⓒ. This is how the interrupts would be serviced:



The important concept to understand is that interrupt priorities determine which device receives the interrupt acknowledge IACK when more than one device is simultaneously requesting an interrupt via IREQ.

Interrupt priorities have nothing to do with whether Ⓑ can be interrupted by Device A once Ⓒ has started executing. Device A has lower interrupt priority than Device C; however, once Device C's interrupt request has been acknowledged, Device C removes its interrupt request. Device A's interrupt request is still present and is the highest priority interrupt request. The instant program Ⓒ enables interrupt logic, it will immediately be interrupted, and program Ⓐ will execute. If you do not want program Ⓒ to be interrupted, then when you write program Ⓒ you must make sure it keeps interrupt logic disable. Instruction steps to do this are described in Chapter 7.

INTERRUPT PRIORITY AND MULTIPLE REQUEST LINES

Let us now return to the Interrupt Priority device illustrated in Figure 5-13, and explain how this device works. As illustrated, there are 16 separate and distinct lines via which external devices can transmit interrupt request (IREQ) signals to the Interrupt Priority device. Signals terminate at pins I0 through I15. I0 has highest priority, while I15 has lowest priority.

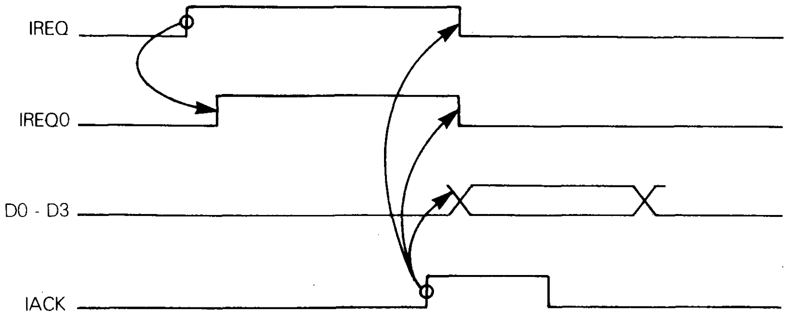
When one or more interrupt requests arrive at pins I0 - I15, logic within the Interrupt Priority device sets IREQ0 high.

At some future time the CPU responds by setting IACK high.

When the Interrupt Priority device receives IACK high, it sets IREQ0 low again, then places the lowest interrupt request line number on D0 - D3.

For example, if one IREQ arrives at I5, 0101 is output via D0 - D3 when IACK is sensed high. If two IREQ signals arrive, at pins I6 and I13, 0110 is output at pins D0 - D3 when IACK is sensed high.

Timing is as follows:



Once the interrupt has been acknowledged, the Data, I/O, READ and WRITE signals are used to transmit data to and from external logic. The data lines interface the Interrupt Priority device to the External System Bus, while the I/O pins interface the Interrupt Priority device with external devices, as illustrated in Figure 5-14.

External devices can use an Interrupt Priority device to initially request an interrupt, but once the interrupt has been acknowledged, the external device can transfer data to or from the microcomputer system via the data lines of the External System Bus. Of course, this means that the external device is selected via memory addresses, as described earlier in this chapter.

The important point to note is that any programs executed after an interrupt has been acknowledged use the same logic as programs executed before the interrupt was acknowledged. Interrupt logic applies only to the process of requesting and accepting an interrupt. If external devices connect to the microcomputer system via I/O ports before the interrupt, they will do likewise after the interrupt has been acknowledged. If external devices connect to the microcomputer system via the data lines of the External System Bus, they will do so before and after an interrupt is acknowledged.

INTERRUPT PRIORITY AND DAISY CHAINING

If a microcomputer system only has one interrupt line, or if there is more than one external device using the same priority interrupt request, then a method called "daisy chaining" must be employed to determine interrupt priorities.

A number of devices in a daisy chain will all connect to the same interrupt request line IREQ. The interrupt acknowledge line, IACK, however, will terminate at one external device. This device must have internal logic which passes on the interrupt acknowledge if the device is not requesting an interrupt, but traps it otherwise. Each external device in the daisy chain contains this same

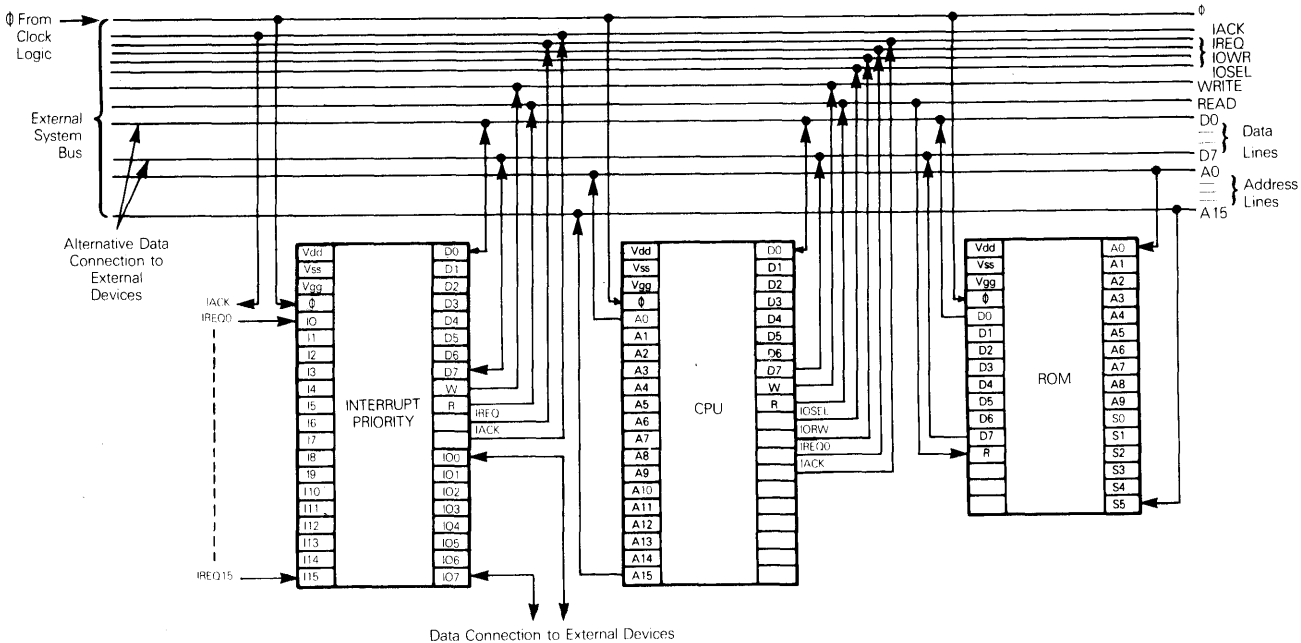
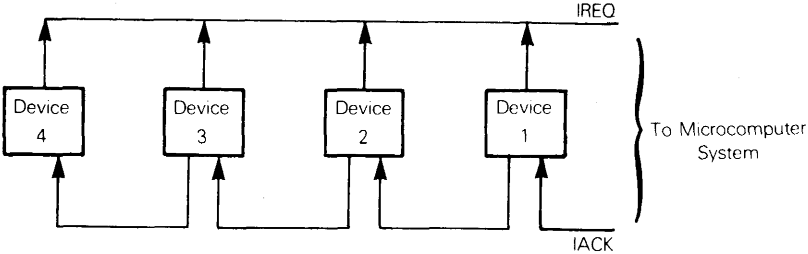


Figure 5-14. An Interrupt Priority Device Connected To the External System Bus

logic, except for the last device, which has nowhere to pass the acknowledge on to. Daisy chaining may be illustrated as follows:



There are strengths and weaknesses associated with having separate priority lines, or daisy chains, in microcomputer systems. In either case, the question becomes somewhat academic since, as described in Chapter 6, a microcomputer system that is being interrupted by a great multitude of external devices is probably being misused. Nevertheless, let us consider some of the strengths and weaknesses associated with separate interrupt priorities, and with daisy chains.

There are situations where separate interrupt priority makes sense, because the microcomputer CPU must attend to one external condition at the expense of all others.

For example, most minicomputers have a highest priority interrupt which is activated by a power failure. While this may not at first make a lot of sense, consider what happens when power does go down.

POWER FAIL INTERRUPT

Microcomputers typically use +5 and/or +12 volt DC power supplies which are generated from the normal 110 volt AC power line. Power failure might be detected when power falls below 90 volts. But it may be a few thousandths of a second before power drops so low that +5 and +12 volts cannot be maintained for the microcomputer. In these few milliseconds, a hundred or more microcomputer instructions may be executed to prepare for power failure in an orderly fashion; now when power comes up again, the power fail interrupted program can restart without loss of data.

We have described one situation where separate interrupt priority lines make sense. **Next consider the limits of daisy chaining.**

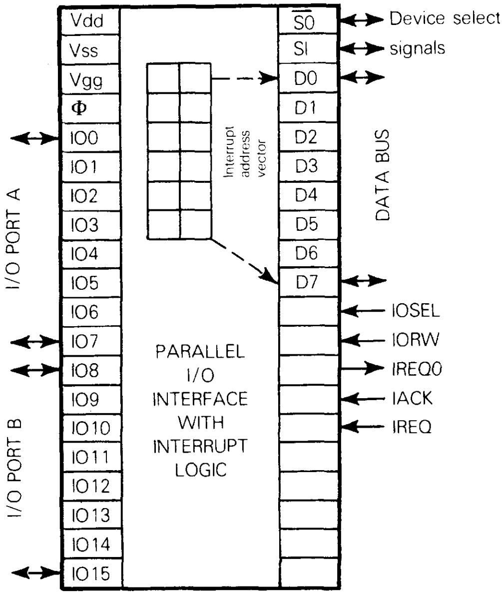
Daisy chaining will handle a number of devices, all of which require interrupt service, so long as the number does not become too large. Consider how little service the 100th shower would get if the microcomputer system must first respond to the needs of the 99 showers that came before it in the daisy chain. It is quite conceivable that the microcomputer system will be so busy attending to devices situated at the beginning of the daisy chain that the tail-end devices would get little or no attention. The occupant of Room 100 will get scalded — or freeze.

Another problem with daisy chaining is that it demands intelligence of any external device in the daisy chain. Once again, we are dealing with microcomputer economics. External devices in a daisy chain must identify themselves, otherwise the microcomputer system has no way of knowing how far down the daisy chain the interrupt acknowledge signal IACK went before it got trapped. So we are back to demanding that external devices in the daisy chain contain sufficient logic to trap the acknowledge signal when an interrupt is being requested, then to transmit a device identification code to the microcomputer system. Certainly this logic could be implemented for a few dollars, but remember a microcomputer does not cost too many dollars either.

DAISY CHAINING WITH I/O INTERFACE DEVICES

In order to eliminate the cost of external logic required to implement a daisy chain, some microcomputer manufacturers provide this logic on support devices. Consider daisy chain logic on an I/O interface device;

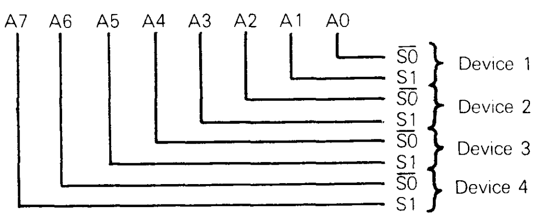
Here is one possible Parallel I/O Interface device configuration:



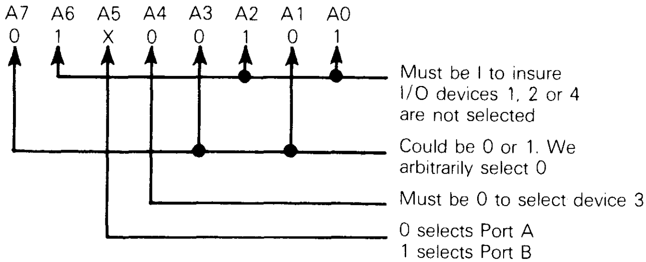
In order to acquire the extra pins for needed signals, we have replaced 8 device select pins (A0 - A7 in Figure 5-12) with two pins, S̄0 and S1. For the device to be selected, a low signal must be input at S̄0. A simultaneous low input at S1 will select Port A. A simultaneous high input at S1 will select Port B.

DEVICE SELECT LOGIC

You can usually choose two address lines and tie them directly to S̄0 and S1, thus selecting individual Parallel I/O devices without using a lot of external device select logic. This is how you could use the 8 address lines illustrated in Figure 5-12 to select four Parallel I/O devices:



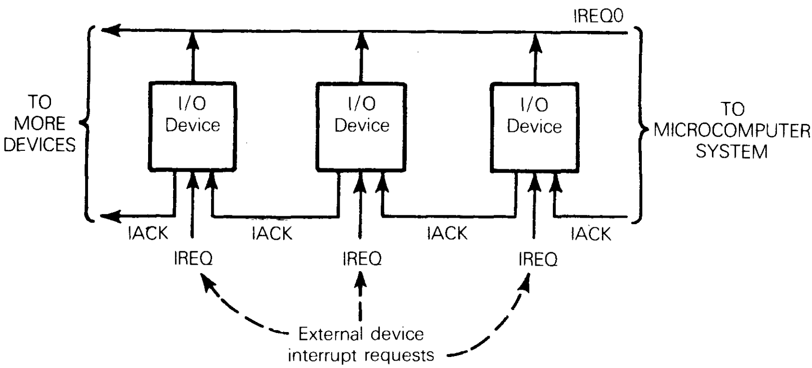
What addresses would be assigned to I/O ports A and B of Device 3? You can compute addresses as follows:



Thus 45_{16} selects device 3, Port A.
65_{16} selects device 3, Port B.

This device select logic has nothing to do with interrupts — which is the subject we are currently discussing.

A number of Parallel I/O Interface devices with interrupt logic would be used as follows:



This is what happens:

1) When one or more external devices request an interrupt, they set their IREQ line "true".
2) When one or more Parallel I/O devices receive a "true" IREQ, they pass it on to the CPU via the common IREQ0 line.
3) The CPU receives a "true" IREQ0 input. CPU logic now knows that some device — it does not know which — is requesting an interrupt.
4) When CPU logic allows the interrupt to be acknowledged it outputs IACK "true".
5) The first Parallel I/O device in the daisy chain receives IACK "true". If it has received IREQ "true", it traps the IACK signal, without regard to whether or not it has been selected via the S0 and S1 select lines. If the first Parallel I/O device in the daisy chain has a "false" IREQ input, it passes the IACK signal on to I/O device 2. The first I/O device with IREQ input "true" traps IACK.

6) The I/O device which traps IACK now outputs its interrupt address vector on the data bus; this allows the I/O device to be identified. Note that select logic associated with $\overline{S0}$ and S1 is still not involved. The interrupt request/acknowledge logic has its own select logic — it is part of the logic which traps IACK or passes it on.

7) Now the interrupt service program takes over. If data is to be input or output, $\overline{S0}$ and S1 are used to select the appropriate I/O device, just as any other program would do.

MULTI-FUNCTION DEVICES

The Parallel I/O with interrupt logic device introduces a very important concept — putting more than one function on a single chip. Parallel I/O logic and Interrupt logic have nothing in common. They happen to be on the same chip, so they share the data bus pins D0 - D7.

External logic can request an interrupt via one I/O device; then after the interrupt has been acknowledged, the external logic can transmit or receive data via I/O ports of the same I/O device, another I/O device, or via no I/O device.

Following an interrupt, there is no reason why you must transfer data via the I/O ports of the I/O device which trapped IACK.

I/O logic and interrupt logic happen to share a chip; they have no other connection.

DIRECT MEMORY ACCESS

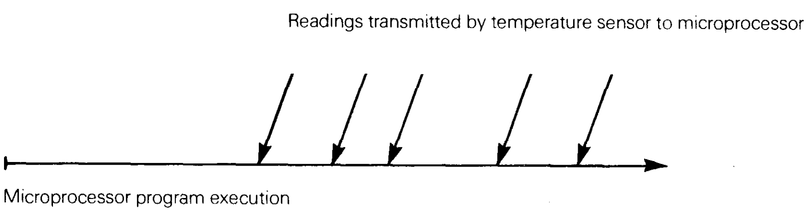
The shower temperature controlling microcomputer system will spend a lot of its time simply receiving data from the temperature sensor and storing the data in a RAM buffer.

We have described how interrupts may be used to execute program RECORD whenever the temperature sensor is ready to transmit data. Let us take another, more careful look at this scheme.

Remember the temperature sensor can transmit approximately two temperature readings per second. To the microcomputer system, this is equivalent to receiving a temperature reading once every quarter of a million instruction executions — approximately.

ASYNCHRONOUS EVENTS

A cheap temperature sensor is not going to transmit exactly two temperature readings per second. In fact, there could be considerable time period variations between temperature transmittals. As a result, we cannot predict, with any degree of accuracy, the time delay between consecutive data transmittals from the temperature sensor to the microcomputer system. Therefore, data transmittals from the temperature sensor to the microcomputer system constitute asynchronous events:



Because data transmittals from the temperature sensor to the microcomputer system are somewhat unpredictable (or asynchronous), program RECORD must be executed every time the temperature sensor transmits a data item. Program RECORD contains the instruction sequence which will move data from an I/O port to a byte in the RAM memory buffer; this instruction sequence

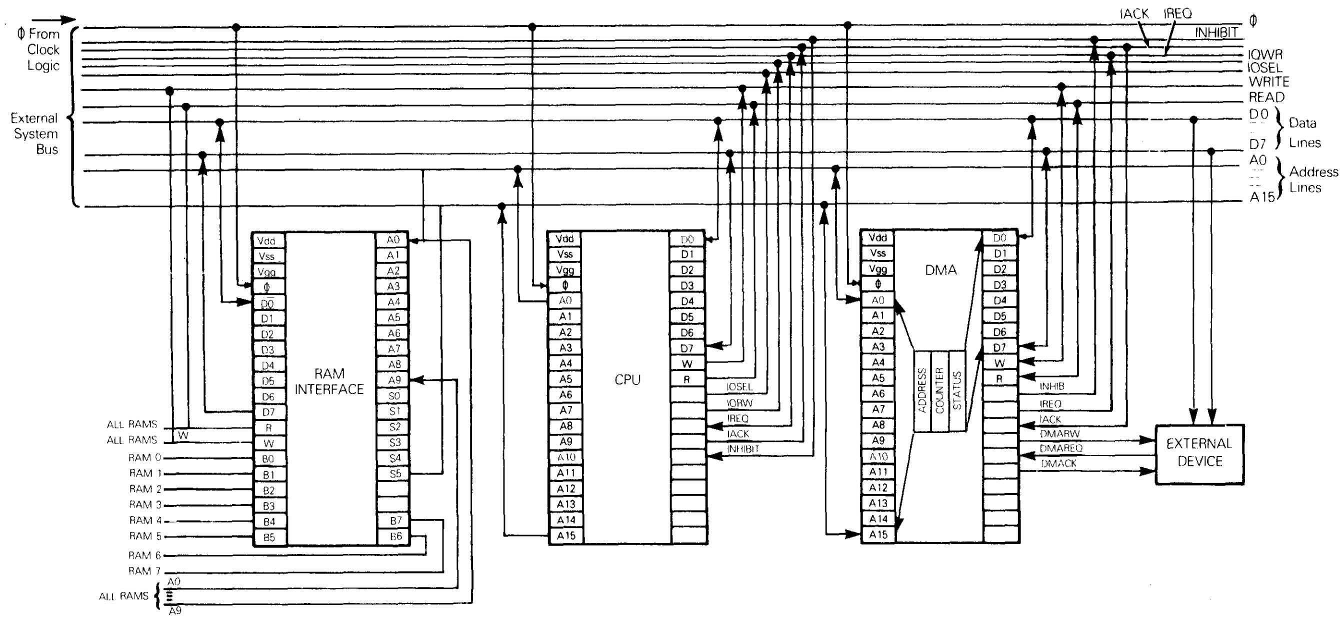
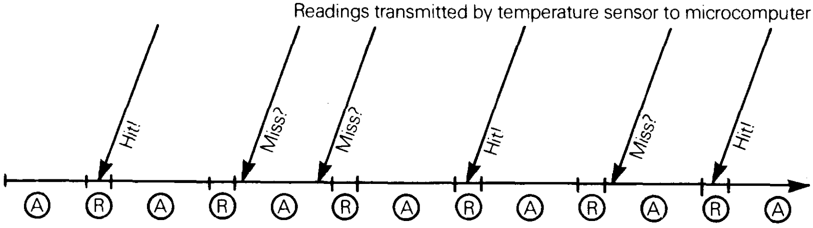


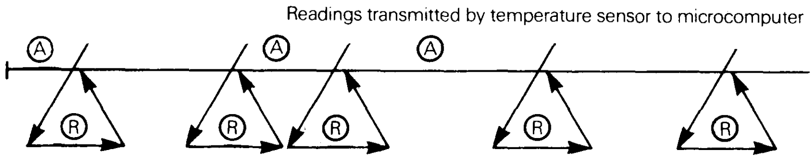
Figure 5-15. Cycle Stealing Direct Memory Access

cannot be part of program ADJUST, since the logic of program ADJUST cannot detect the arrival of data from a temperature sensor. **Any scheme that executes the RECORD instruction execution sequence at fixed time intervals is bound to miss a large number of the data transmittals from the temperature sensor.** Here is an example:



Ⓐ represents normal execution of program ADJUST. Ⓡ represents fixed, periodic execution of the instruction sequence which records data transmitted by the temperature sensor to the microcomputer system. If transmitted data does not reach the microcomputer system during Ⓡ it may be missed.

The only safe way of catching all data transmitted by the temperature sensor is to have the temperature sensor request an interrupt when it is ready to transmit a data item. In response to the temperature sensor's interrupt request, the microcomputer system will execute the data sensing instruction sequence, characterized in the illustration above by Ⓡ The illustration must now be modified as follows:



Each time the temperature sensor transmits data to the microcomputer system, it notifies the CPU by requesting an interrupt. In response to the interrupt request, program logic suspends execution of program ADJUST (Ⓐ) while executing program RECORD (Ⓡ). Program RECORD reads the data input by the temperature sensor, then program ADJUST continues executing from the point of suspension.

Even this method of recording data transmitted by the temperature sensor is not very efficient. **This is what happens when the CPU accepts an interrupt and executes RECORD:**

1) The CPU is executing program ADJUST (Ⓐ) in the illustration above. When the CPU senses an interrupt request, it executes some instructions which save the contents of CPU registers and status; then it executes an instruction to acknowledge the interrupt.

2) Program RECORD (Ⓡ in the illustration above) is executed. This program contains instructions which load a memory address into the Data Counter, read data from an I/O port into the Accumulator, then output the data from the Accumulator to the memory word addressed by the Data Counter.

3) Step 1 is reversed. Saved contents of registers and status are restored and program ADJUST continues execution.

Out of all the instructions that get executed to implement the above three steps, the only change, each time the temperature sensor transmits a reading and Ⓡ is re-executed, is the contents of the Data Counter in Step 2. The contents of the Data

Counter will be one more than it was last time, and one less than it will be the next time. Fifty microseconds or more are needed to repetitively process an otherwise predictable and trivial sequence of events. Before we can decide whether this is a serious or an inconsequential problem, we must ask two questions:

1) **Are operations like this fairly common, or is this an isolated and special situation?**

   The answer is that it is one of the most common operations performed by a microcomputer. In fact, not only will a microcomputer spend a great deal of time reading data from an external device, it will spend almost as much time routinely transmitting data from RAM buffers to external devices.

2) **If the microcomputer did not spend 50 microseconds every time it input data from an external device (or output to an external device), what else would it do with the time?**

   In many simple applications the answer is nothing, in which case the wasted time is irrelevant. But clearly, as a microcomputer application starts to get more complex, the waste of time starts to become more serious. If it takes 50 microseconds to read a data item from an external device, and another 50 microseconds to transmit a data item to the same external device, then one microcomputer system could perform one hundred data transfers per second — but there would be no time left to do anything else.

We must therefore conclude that there will be a significant number of applications in which the time wasted processing interrupts will be intolerable.

CYCLE STEALING DIRECT MEMORY ACCESS

Direct memory access (DMA) provides a solution. We will create a new device for our microcomputer system, and on this device we will place a small amount of CPU-type logic, dedicated to the sole task of moving data between I/O ports (or the data lines of the External System Bus) and memory.

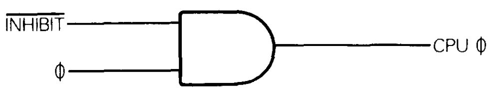
The DMA device will accomplish its task by suppressing or bypassing CPU logic while it creates signal sequences that enable the appropriate data transfer.

Given the microcomputer system architecture that has been described in this chapter, the task of designing a DMA device is really quite straightforward.

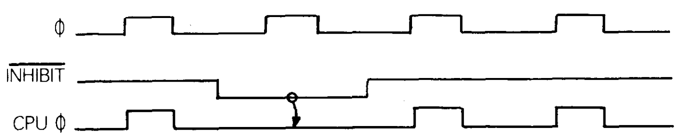
INHIBIT CONTROL

Temporarily suppressing CPU logic is easy, since the CPU is driven by an external clock signal. If we can stop the clock signal, we can stop the CPU. Therefore, as illustrated in Figure 5-15, we will add an INHIBIT control line to our External System Bus, and an INHIBIT control pin at the CPU device.

The $\overline{\text{INHIBIT}}$ signal will normally be high, so that a simple AND-gate within the CPU chip will combine the clock signal with the $\overline{\text{INHIBIT}}$ signal, to generate the internal clock signal which drives CPU chip logic:



All the DMA device has to do is set the $\overline{\text{INHIBIT}}$ signal low in order to suppress timing clock signals within the CPU chip:



Memory devices and Parallel Interface devices have no way of knowing where the control signals that drive their logic come from. Therefore, the DMA device can take control of the External System Bus while CPU logic is suppressed.

Consider Figure 5-15 in more detail. The DMA device, as illustrated, contains the following three registers:

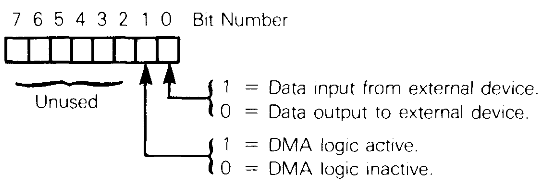
1) **An Address register** which contains the address of the next memory word to be accessed, either for a read or a write operation.
2) **A Counter**, which initially contains the length of the data buffer which is to be filled during a read operation, or out of which data is to be read during a write operation.
3) **A Status register**, which identifies the direction of data flow, whether the DMA logic is currently active or inactive, and various other DMA options. We will look at a few of these options later.

To the CPU, the registers of the DMA device may appear as I/O ports or addressable memory words. Initially, programmed I/O instructions will be used to set values in the Address, Counter and Status registers. Here is an example:

0080	Address Register
007F	Counter Register
03	Status Register

This simple example specifies that a data buffer, $7F_{16}$ bytes long and origined at 0080_{16}, is to be filled with data from an external device, using Direct Memory Access.

The Status register is for the moment limited to this simple format:



By setting a value of 03_{16} in the Status register, data input from an external device is specified, and DMA logic is activated.

DMA INITIALIZATION

A program must be executed by the CPU to initialize a DMA operation. The program will have to load data appropriately into DMA device registers. There is no other way for data to get into the DMA device Address, Counter and Status registers.

In order to initialize a DMA operation, a program would be executed by the CPU to perform these steps:

1) Transmit the low order half of the starting address to the DMA device.
2) Transmit the high order half of the starting address to the DMA device.
3) Transmit the buffer length to the DMA device.
4) Transmit a control code (03_{16}, as illustrated above) to the Status register of the DMA device. The control code must identify the direction of the data transfer and must turn the DMA device on.

Notice that the DMA device has READ and WRITE control lines.

The WRITE control line will be used by the DMA device to write data into RAM.

The READ control line will be used by the DMA device to output data from RAM or ROM.

DMA CAUGHT ON THE FLY

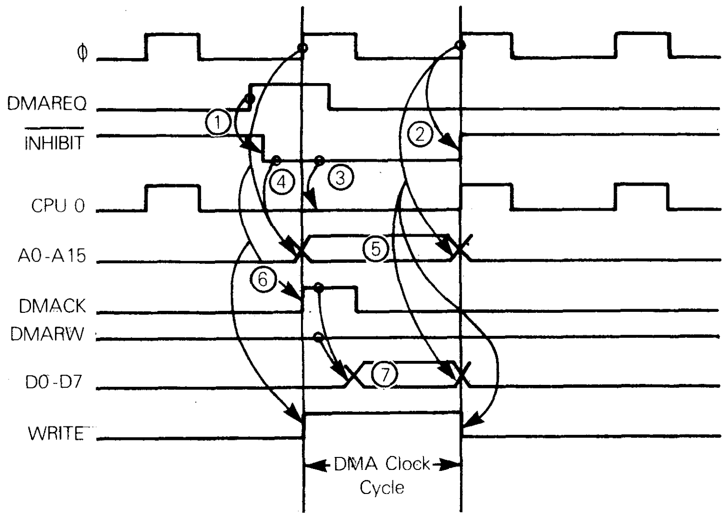
The CPU can, at any time, read the contents of DMA device registers. This allows a program to check on how far a DMA operation has progressed by reading the Address register and/or Counter register contents. **A program that adjusts its logic to the current level of completion of a DMA operation is said to be catching DMA on the fly.**

DMA BEING TURNED OFF

Of course, **an executing program can turn a DMA operation off at any time by simply writing new data into the DMA device's Status register.** As described above, setting the 1 bit of the Status register to 0 would immediately stop any DMA data transfer that was in progress.

DMA EXECUTION

What happens after a DMA operation has started? Notice that external devices are connected directly to the data lines of the External Data Bus. In addition, the external device has a DMA request signal (DMAREQ) which it pulses high to the DMA device. If data is being read from the external device, the following signal sequence occurs:



The entire DMA operation occurs in one clock cycle.

Events occur as follows:

1) As soon as the DMA device senses a high pulse on the DMAREQ line, it immediately lowers the $\overline{\text{INHIBIT}}$ control signal (① above). The $\overline{\text{INHIBIT}}$ control is kept low until the second rising edge of clock pulse ϕ to follow (② above).

FLOATING BUSSES

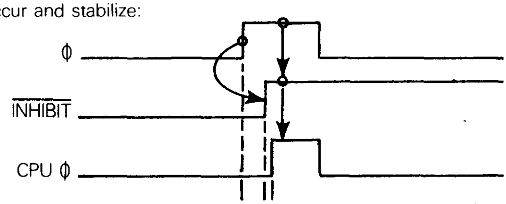
2) The CPU will suspend operations for one clock cycle because $\overline{\text{INHIBIT}}$ keeps the CPU ϕ line low (③ above). The CPU will also apply high resistances at its connections to the address and data busses, effectively disconnecting itself from these busses. This is referred to as floating the busses.

3) The combination of $\overline{\text{INHIBIT}}$ low and Φ rising (④ above) causes the DMA device to output the contents of its Address register on the address lines of the External System Bus (⑤ above). Also the DMA device acknowledges the DMA request by pulsing the DMACK response control line high (⑥ above).

4) The 0 bit of the DMA Status register determines whether the DMARW control line will be high or low. As illustrated above, it is low, indicating to external logic that it must transmit data to the data lines of the External System Bus. The combination of DMACK high and DMARW low causes the external device to place its data on the data lines of the External System Bus (⑦ above).

5) The 0 bit of the DMA Status register also causes the DMA device to output a high on the WRITE control line. All RAM interface devices will decode the address on the address lines and one will find itself selected; on sensing the WRITE control high, the selected RAM interface device will take whatever data is on the data lines of the External System Bus and will write this data into the addressed memory word. RAM interface logic neither knows nor cares where data, address and control signals originated. It simply responds to any situation which activates its internal logic.

6) As illustrated above, the second rising edge of ϕ terminates DMA operations. In reality, some other scheme would be used, since the one illustrated, though very simple, would cause jitter in the leading edge of CPU ϕ. CPU ϕ goes high only because $\overline{\text{INHIBIT}}$ goes high. $\overline{\text{INHIBIT}}$ goes high because ϕ just went high for a second time. Clearly, ϕ going high for a second time cannot exactly coincide with CPU ϕ going high, because in the middle, $\overline{\text{INHIBIT}}$ going high had to occur and stabilize:

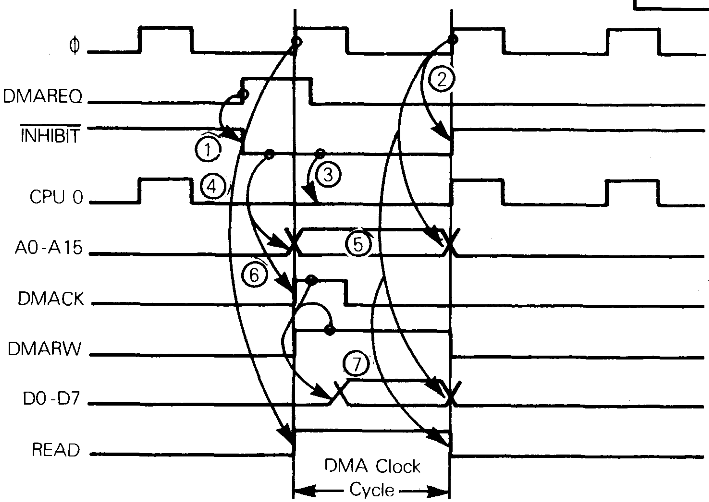


Thus, as illustrated, the DMA read timing diagram is conceptually accurate but realistically impractical.

As soon as the DMA device has output the contents of its Address register to the External Data Bus, it will increment the contents of the register, so as to address the next word of the RAM data buffer. Simultaneously the DMA chip will decrement the contents of its Counter register.

Here is the signal sequence for a DMA write operation:

DMA WRITE TIMING



Notice that the DMA write operation differs from the DMA read operation only in that the DMA device sets the READ control signal high when it places the contents of its Address register on the address lines of the External Data Bus. This causes the memory module which is selected by the memory address to place the contents of the addressed memory word on the data lines of the External Data Bus. DMARW, set high by the DMA chip, causes the external device to read the contents of the data bus.

Once again the DMA device will decrement its Counter register and increment its Address register, so that it is ready for the next DMA operation.

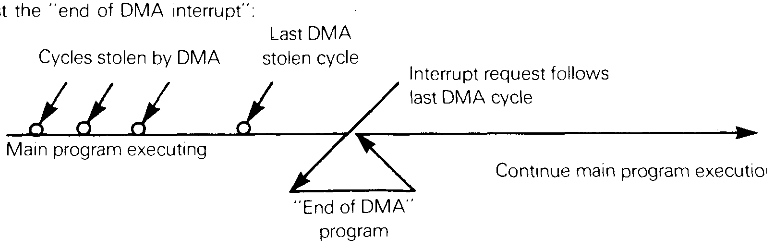
One of two things may happen when the counter register decrements to zero:

DMA END

1) **The DMA device may signal the fact that the DMA operation is over by sending an interrupt request to the CPU.** This interrupt would be handled according to whatever interrupt processing logic the CPU is using.

2) **The DMA device may simply start the whole process over again**, by saving the original value of the Counter register and Address register, so that it can reload these original values and allow operations to proceed endlessly, or until stopped by the CPU.

The two options available when the DMA device counts down to zero are illustrated as follows: First the "end of DMA interrupt":



Next consider a DMA device with additional storage registers for the initial address and buffer length count:

Initial Address
Initial Count

Current Address
Current Count
Status

When "Current Count" goes to 0, "Initial Count" is loaded into "Current Count", and "Initial Address" is loaded into "Current Address"; Status remains unchanged (unless modified by the CPU), and the DMA operation starts over, re-accessing the same buffer, from the same beginning.

DMA WITH MULTIPLE EXTERNAL DEVICES

One DMA device can control DMA operations for many external devices. Since a DMA device does not actually transfer any data, it does not need any I/O ports through which external devices communicate with the microcomputer system. The external devices connect directly to the data lines of the External System Bus. This arrangement is very convenient, since it allows the DMA device to control DMA access for a number of external devices.

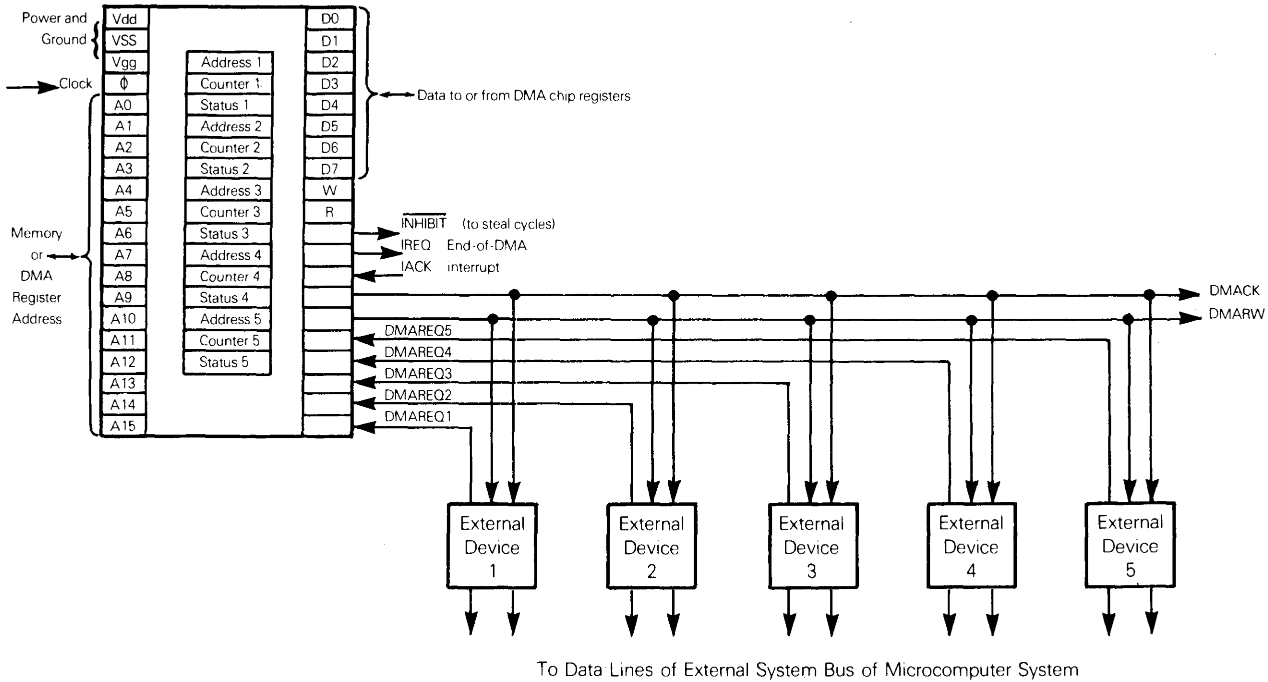


Figure 5-16. DMA Device Controlling DMA Operations For Five External Devices

Many schemes could be devised which allow one DMA device to control DMA access for more than one external device; Figure 5-16 illustrates just one possibility.

The DMA device illustrated in Figure 5-16 controls DMA access for five external devices. Each external device has its own DMA request line (DMAREQ1 through DMAREQ5). Common DMA acknowledge (DMACK) and DMA read-write control (DMARW) lines are used by all devices.

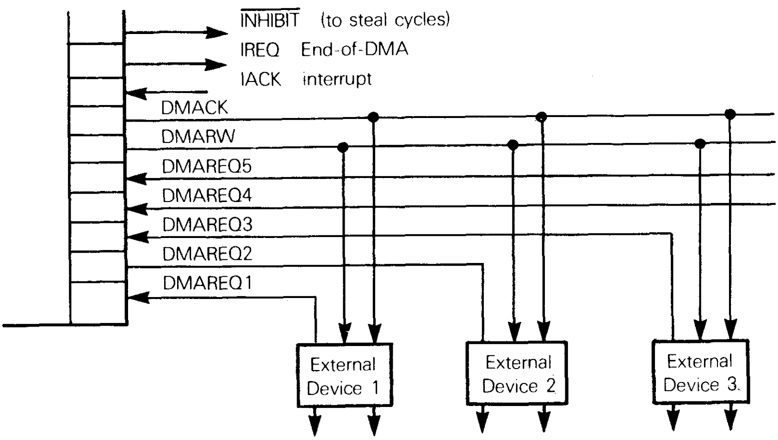
There are five sets of registers within the DMA device, one set for each external device. Each external device, when it is ready to transmit or receive data, indicates this fact by setting its own DMA request line high. The DMA device logic accesses the correct three data registers, based on the DMA request received. For example, DMAREQ3 high identifies Address 3, Counter 3 and Status 3 as the registers containing the data to be used this time.

When the multidevice DMA device illustrated in Figure 5-16 steals a cycle and transfers a data word via DMA, the signal sequence is identical to that which we have already described for a single external device. However, external devices in Figure 5-16 must contain their own select logic. In other words, a device which raises its DMA request line must be the only device to respond to DMACK, DMARW and the External Data Bus; no other device attached to the multidevice DMA device must respond to these lines. It is the responsibility of the external device, not the multidevice DMA device, to ensure that only one external device considers itself selected at any time.

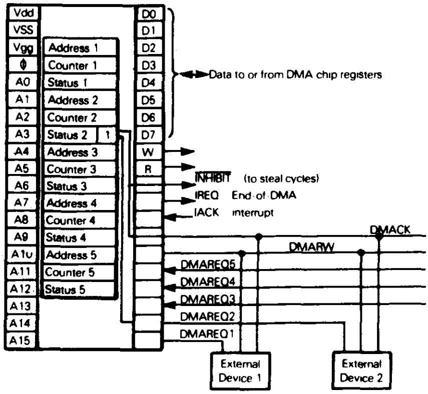
So long as only one external device considers itself selected for DMA at any time, then there is no possibility of confusion in DMA data transfers. Memory modules merely respond to address and control signals output by the DMA device. Memory modules neither know nor care where this information had its origin. Only the selected external device is active at the other end of the External System Bus, so the two ends of the data transfer are clearly defined.

Let us consider an example in detail.

External device 2 is ready for another data access, so it raises DMAREQ2 high:

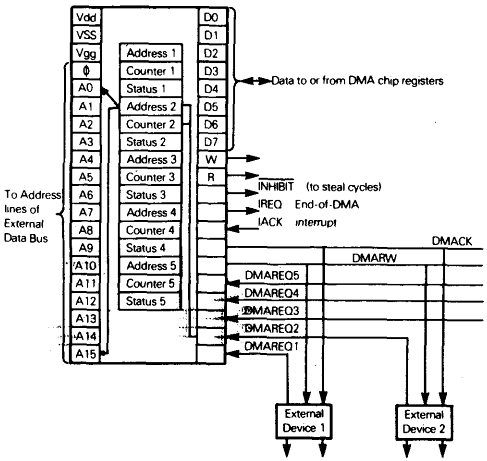


The first thing DMA device logic will do is check the status register associated with DMAREQ2, in this case, register Status 2. If the enable bit (bit 0) is 0, DMA device logic will ignore the DMA request. Since DMAREQ2 is a pulse signal, it will go away. If the enable bit is 1, DMA device logic will acknowledge the interrupt on DMACK and steal a CPU cycle by lowering $\overline{INHIBIT}$:



For the upcoming DMA operation, only external device 2 can participate at the external end of the data transfer.

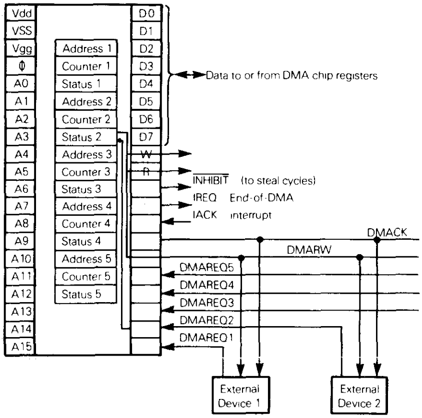
DMAREQ2 causes the contents of Address register 2 to be output at the address pins:



DMA device logic will decrement the contents of Counter 2 and increment the contents of Address 2.

Since DMAREQ2 identifies Address 2 as the DMA register containing the required memory address, no confusion can result from the fact that four other addresses, in four other address registers, are present.

DMAREQ2 also identifies Status 2 as the Status register controlling current operations. The W, R and DMARW control lines are set based on the contents of Status 2:



A data transfer now occurs between the memory word addressed by Address 2 and external device 2. Signal sequences associated with the data transfer are exactly as described for the single external device DMA chip.

SIMULTANEOUS DMA

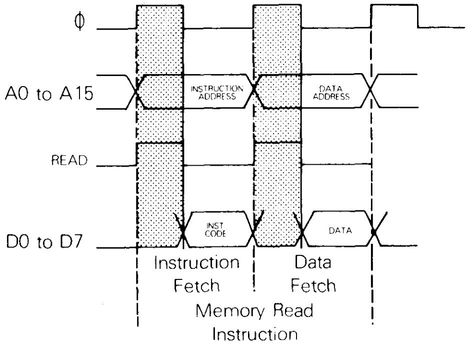
Observe that cycle stealing DMA really involves nothing more than reproducing a limited amount of the CPU logic on a DMA device. **A DMA device may be likened to a CPU which is capable of executing just two instructions:**

1) Transfer data from an external device to memory.

2) Transfer data from memory to an external device.

Because of the very limited number of operations which the DMA device can perform, nearly all of the time-consuming sequences associated with CPU operations (for example the instruction fetch) can be eliminated.

But we can take DMA logic a step further. By duplicating part of the External System Bus, we can eliminate the need to steal cycles from the CPU. Look again at the timing diagram for a memory read operation:



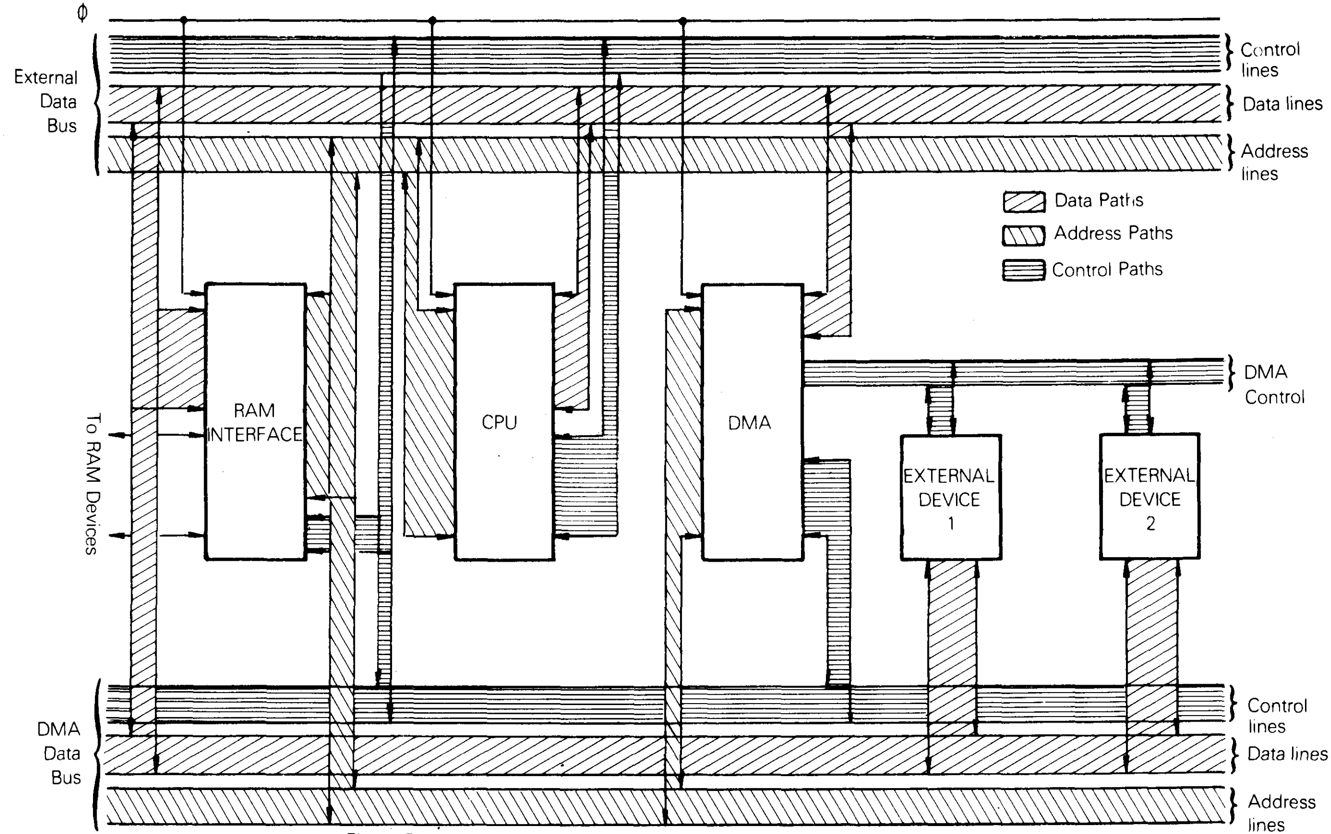


Figure 5-17. Data, Address and Control Paths Used In Simultaneous DMA

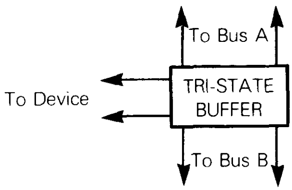
Notice that even though the External System Bus is constantly busy, neither memory nor external devices will be busy during the Φ high portion of instruction cycles. These periods, shaded above, represent the time required by the CPU's Control Unit to generate appropriate control signals.

By creating a second External System Bus, DMA logic can access memory modules while the CPU is not doing so. This is illustrated in Figure 5-17.

External devices will be connected to the data lines of the DMA System Bus, as shown in Figure 5-17, for all DMA data transfers. If the external devices also access the microcomputer system using programmed I/O, then there must be additional connections to the data lines of the External System Bus; these additional connections are not shown in Figure 5-17.

TRI-STATE BUFFER

At the memory modules, the same memory and address pins must communicate with two busses and that will require some form of T-junction. This T-junction is referred to as a "tri-state" buffer. A tri-state buffer, is, in effect, nothing but a multiple signal T-junction:



SIMULTANEOUS VERSUS CYCLE STEALING DMA

What about the economics of Simultaneous DMA? We must pay for another External Data Bus and a number of tri-state buffers. What we buy is a little time: one clock cycle for every byte of data transferred.

In reality, the extra cost for the tri-state buffer is very small; in fact, it is not inconceivable that memory modules will be provided with tri-state buffers built into them. We are therefore talking about a very small additional expense for a very small performance improvement.

A microcomputer designer is therefore more likely to select one DMA method or the other based upon which method is best suited to the architecture of the microcomputer system.

THE EXTERNAL SYSTEM BUS

The signals on the External System Bus, as illustrated throughout this chapter, do not define a standard configuration to which all microcomputers must conform.

The External System Bus represents one of the most varying features of any microcomputer system. Indeed, the only constant feature you will find from bus to bus is the presence of eight data lines (for 8-bit microcomputers). Most microcomputer systems will also have 16 address lines.

The greatest variation is seen in the control signals generated by the CPU. Basically there are two philosophical extremes. One extreme calls for a complex set of control signals to which other devices passively respond. The other extreme calls for elementary control signals which must be interpreted by devices that contain a considerable amount of internal chip logic.

Consider first the CPU that totally dominates a microcomputer system. Devices that interface to this CPU will not receive any clock signal inputs. Instead they will receive numerous control signals which identify events on the Data Bus in detail. The National Semiconductor and Signetics microcomputers are the best examples of this philosophy.

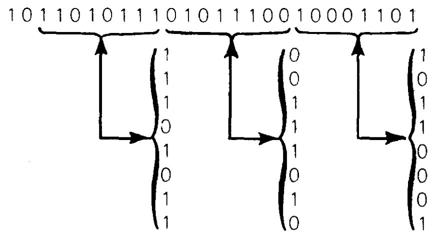
The MCS6500 represents the other extreme. This microcomputer outputs just one control signal to identify either data input or output. All devices that support the MCS6500 CPU receive this control signal, plus the system clock signal. Devices contain internal logic to decode this combination of two signals according to the rules of the MCS6500 microcomputer system.

The philosophy behind National Semiconductor and Signetics type microcomputers is that no special devices are needed to support the CPU. By having a very complete set of control signals, standard off-the-shelf logic can be used.

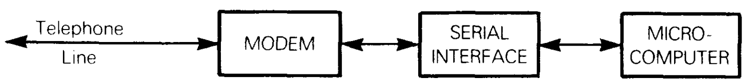
The philosophy of microcomputers with very elementary sets of control signals is that since it costs virtually nothing to add extra logic to an LSI chip, you are far better off generating devices that support the CPU in a very precisely defined way.

SERIAL INPUT/OUTPUT

Data is transferred over telephone lines serially. There are also some slow I/O devices, such as the common teletype and magnetic tape cassettes, which transmit and receive data serially. If a CPU is to transmit or receive serial data, then it must have interface logic capable of converting serial data to parallel data, or parallel data to serial data:



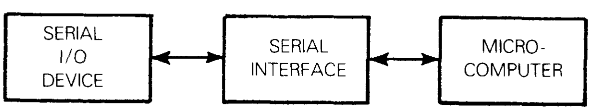
These are the steps via which data is transferred between a telephone line and a microcomputer system:



MODEM

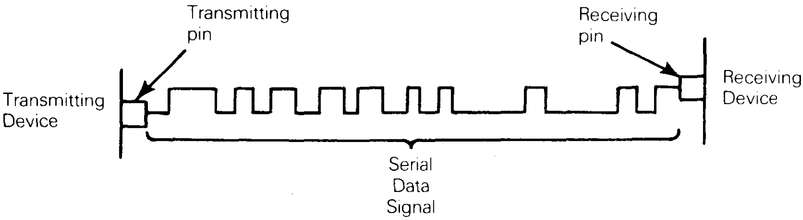
A modem is a device which can translate telephone line signals into digital logic levels, or digital logic levels into telephone line signals. Some microcomputer manufacturers provide modems as a single logic device, but we do not consider the modem to be part of the microcomputer system; therefore **modems are not described in this book.**

A magnetic cassette unit, or teletype, or any other serial device can connect directly to the serial interface:



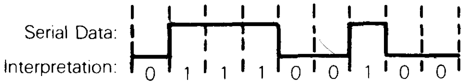
IDENTIFYING SERIAL DATA BITS

The unique property of a serial data stream is that the data is transmitted and received as a single signal, via single device pins:



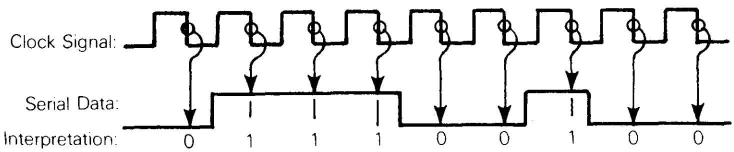
How is the receiving device to interpret a serial signal Like any other digital signal, the data signal can have a "high" level (+ 5v) representing the digit 1, or a "low" level (0v) representing the digit 0.

Consider the binary data sequence: 011100100. This is its serial data signal representation:

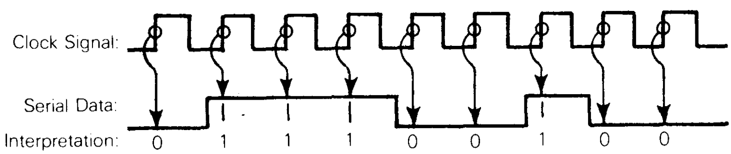


CLOCK SIGNAL

Whereas it is easy for you to look at the serial data signal and interpret it within the vertical broken lines, the receiving device will require more tangible evidence of data bit boundaries. **We will use a clock signal to identify the instant at which the receiving device must interpret the data signal:**

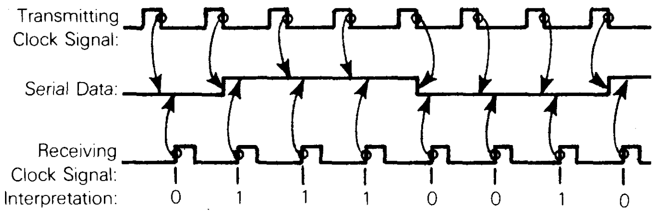


As illustrated above, the falling edge of the clock signal identifies the instant at which the serial data signal must be sampled. We could just as easily clock on the rising edge of the signal:



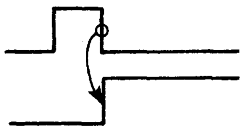
A serial data signal must be created by the transmitting device before it can be interpreted by the receiving device. Let us look into the implications of this simple necessity.

If the receiving device uses a clock to interpret the serial data signal, then the transmitting device must use a clock with the same frequency to create the serial data signal:



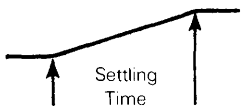
The transmitting and receiving clock signals cannot be identical; this is because in reality, it takes a finite time for a signal to change state.

Now it makes figures easy to follow if signal transitions are drawn as clean square waves:

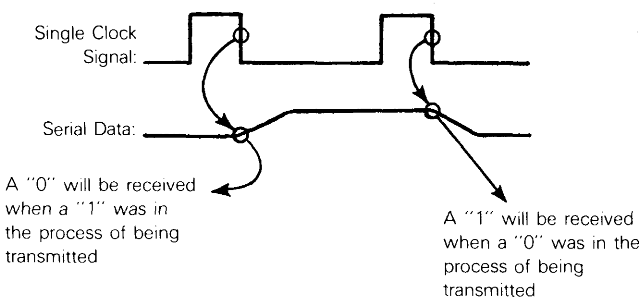


But in reality, every signal that changes state requires a finite settling time:

SIGNAL SETTLING TIME

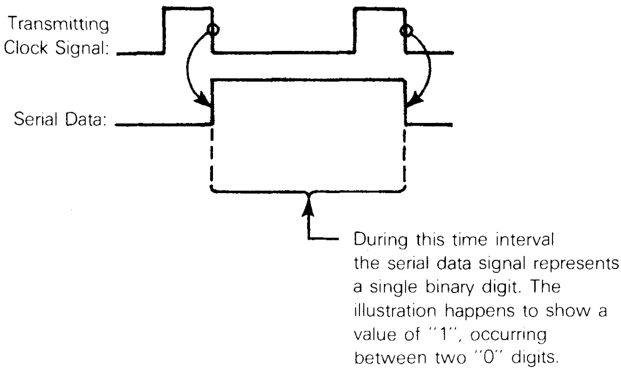


Look at what happens if you use the same signal transition to transmit and receive serial data:



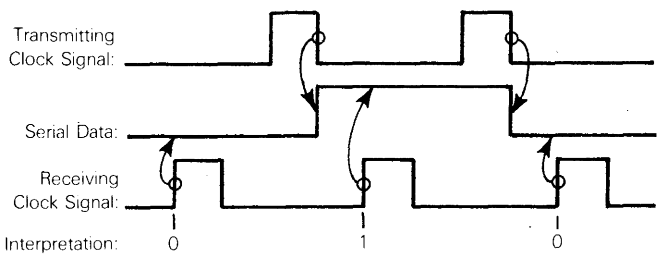
We conclude that **the transmitting and receiving clock signals,** while they have a great deal in common, **cannot be a single signal subject to identical interpretation. The transmitting clock signal will identify the duration of** one binary digit:

SERIAL DATA TRANSMITTING CLOCK SIGNAL



At some point within the single digit time interval a **receive clock signal will identify the serial data signal level:**

SERIAL DATA RECEIVING CLOCK SIGNAL

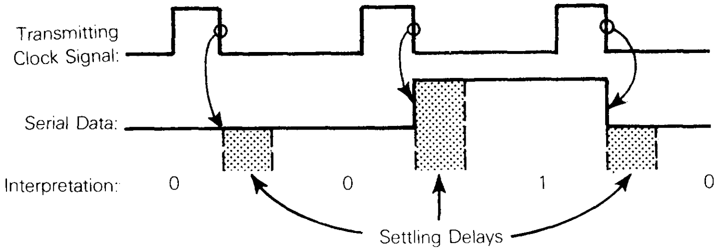


The illustration above shows the transmitting clock signal active on its trailing edge, whereas the receiving clock signal is active on its leading edge; there is nothing significant in this use of signal edges.

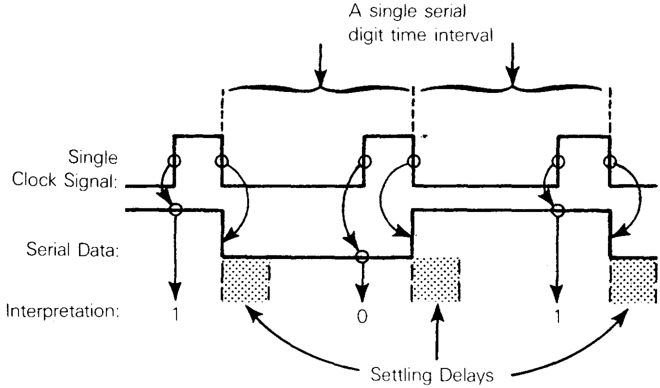
The receiving device must wait for the Serial Data signal to settle, presuming it has changed state, before trying to read the signal level. A signal's settling delay is a characteristic of the

SIGNAL SETTLING DELAY

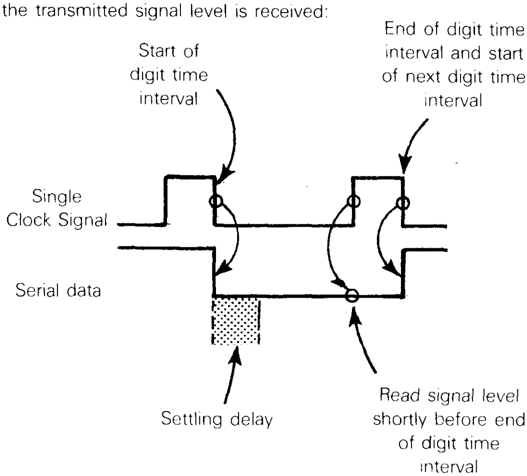
transmitting device; the length of this delay is given by the manufacturer's device data sheet. Here is an illustration of settling delay:



We can use a single clock signal to transmit and receive, providing we transmit on the trailing edge of a clock pulse, and receive on the leading edge of the next clock pulse:



Look carefully at how the transmitted signal level is received:



BAUD RATE

The time interval during which the Serial Data signal represents a single binary digit is directly related to the speed at which data is being transmitted. Suppose 110 digits per second are being transmitted; this is a common transmission speed. Each serial digit will then endure for:

$$\frac{1000000}{110} = 9091 \text{ microseconds}$$

However, the duration of a digit in a serial data stream is not the way in which serial data transfers are measured; instead, **we measure "bits per second", and refer to this number as the BAUD RATE.** For example, if 110 digits per second are transmitted, this is equivalent to a baud rate of 110.

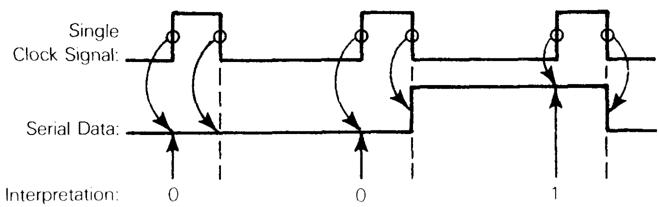
CLOCK SIGNALS

Our microcomputer system already has a clock signal, used to time instruction execution within the CPU. **Do not confuse the microcomputer system clock with the serial data clock;** the only thing these two signals have in common is that they are both clock signals. **The serial data clock signal may or may not be derived from the microcomputer system clock.**

From a microcomputer user's point of view, speed is the most striking difference between the microcomputer system clock and the serial data clock. A typical microcomputer system clock may have a period of 500 nanoseconds (2 MHz) whereas serial data transfer rates typically range from between 110 and 9600 Baud 110 Hz to 9.6 KHz. In other words, the fastest serial data transfer rate is approximately 200 times slower than a typical microcomputer CPU clock.

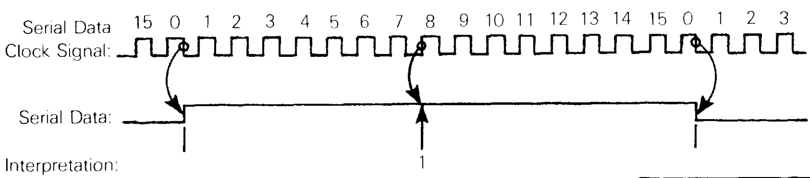
SERIAL x 1 CLOCK SIGNAL

The serial I/O clock does not necessarily have to pulse at exactly the baud rate, although frequently it does:



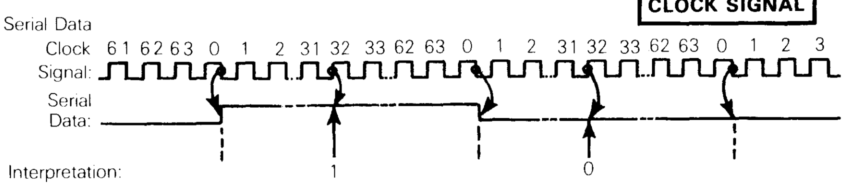
SERIAL x 16 CLOCK SIGNAL

It is quite common for the clock rate to be 16 times the baud rate:



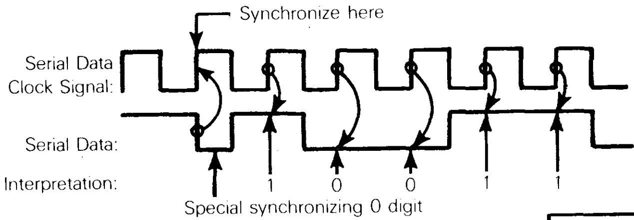
SERIAL x 64 CLOCK SIGNAL

64 times the baud rate is also a frequent option:



The reason for having x16 and x64 clocks is to get as close as possible to the center of a single digit time interval when sampling the serial data signal.

The fact that serial data needs a companion clock signal does not necessarily mean that all serial I/O requires two signal lines. The accompanying clock signal does not actually have to be transmitted on a companion wire. If you set up a serial data communications interface, with a predefined baud rate, then **receiving device logic does not have to receive a companion clock signal.** Receiving device logic can create its own, local clock signal, synchronizing it with a transition in the serial data line:



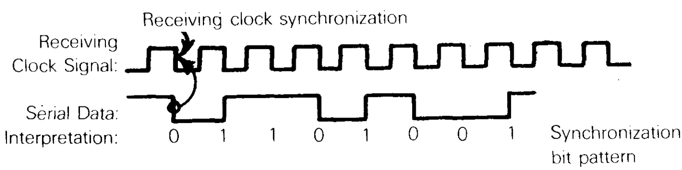
MARKING

In the illustration above, the Serial Data signal is permanently high when not transmitting data; this is often referred to as Marking.

Note that when using a x16 or x64 clock signal, the receive clock can be one or two pulses out of phase with the transmit clock and no harm will be done. The receive sampling point will simply be skewed a little off center.

If a single synchronization binary digit is insufficient, **how about a synchronizing digit pattern?**

We can, for example, define a special synchronization serial data bit sequence and set up rules which state that every serial data stream must be preceded by this synchronization pattern:



SYNC CHARACTER

The synchronization pattern illustrated above does exist, in the form illustrated; and **is referred to a SYNC character.**

PROTOCOL IN SERIAL DATA

Specifying that a serial data stream must use synchronization digits or characters is just the first of **many rules** that we must **impose on serial data streams in order to ensure that the receiving device correctly interprets the transmitted data. This set of rules is referred to as "communications protocol".**

Every serial I/O data link must have a communications protocol, since the serial data must be completely self-defining. Unlike parallel I/O, the serial data line cannot always be accompanied by control lines which tell the receiving device how to interpret the data at any instant.

TELEPHONE LINES

When dealing specifically with telephone lines, consider the fact that the transmitting and receiving devices may continuously switch roles, as happens in any voice telephone conversation. While talking, you are the transmitter; while listening, you are the receiver.

Similarly, when transmitting serial data over telephone lines two-way communication will almost always be required.

If a single telephone line is used to transmit data in both directions then communication is said to be half duplex.

HALF DUPLEX

If two telephone lines connect the transmitting and receiving devices, with each line being dedicated to data transfer in one direction only, **then communication is said to be full duplex.**

FULL DUPLEX

The advantage of full duplex telephone communication is that data transfer in both directions can proceed in parallel.

ERROR DETECTION

Whether serial data is being transmitted over telephone lines, or directly between a transmitting and a receiving device, we must check for errors in transmission.

If spurious data signals find their way into the serial data line, the receiving device must have some means of determining that errors have crept into the data.

At a primitive level, the parity bit does this job. **Since the parity bit has been set or reset, to ensure that the total number of 1 bits in the data unit is either odd or even, then an odd number of error bits will be detected.** Here are some examples, assuming odd parity; in all illustrations, the parity bit is shaded and error bits are starred:

PARITY BIT

Transmitted	Received	
101101100	100101100 *	Even parity, error detected
101101100	110101100 **	Odd parity, no error detected
100101101	100101100 *	Even parity, error detected
100101101	011010011 *******	Odd parity, no error detected

An additional technique used to check for errors in transmission is to append a "cyclic redundancy character" at the end of data stream segments. The cyclic redundancy character is a number created by dividing the transmitted data stream by a fixed polynomial. Here is one commonly used 17 binary digit divisor:

CYCLIC REDUNDANCY CHARACTER

11000000000000101

The result of dividing this divisor into the transmitted data stream, treating the transmitted data stream as one continuous binary number, becomes the Cyclic Redundancy Character. The receiving device multiplies the received data stream by the Cyclic Redundancy Character. If the result is not the standard divisor, then an error must exist.

The Cyclic Redundancy Character is just one rather simple method used to track down errors in transmission. Very complex methods have been devised not only to track down errors, but also to determine exactly what the error is — so that it can be corrected. Entire books have been written on the subject of error detection and correction, therefore we are not going to discuss the subject any further.

SERIAL INPUT/OUTPUT PROTOCOL

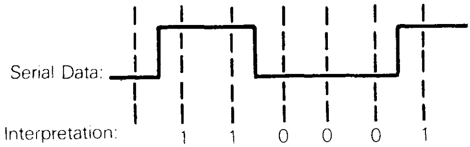
Let us now tie together the miscellaneous necessities of serial data transfer which have been described thus far.

Generally stated, serial data communications protocol can be divided into synchronous and asynchronous categories. You will find protocol easier to understand if you

approach synchronous and asynchronous data communications as two separate and distinct entities — not minor variations of a single concept.

SYNCHRONOUS SERIAL DATA TRANSFER

The principal characteristics of synchronous, serial data transfer is that the data conforms exactly to a clock signal. Having once established a serial data transfer baud rate, the transmitting device MUST transmit a data bit at every clock pulse; therefore the receiving device knows exactly how to interpret the serial data signal:



For example, if 300 baud serial synchronous data transfer has been specified, then the receiving device can slice the serial data signal into 3333 microsecond segments, interpreting each segment as a single binary digit.

We earlier illustrated clock signals with frequencies 1, 16 or 64 times the baud rate. x16 and x64 clock signals could be used with synchronous serial I/O, but in practice they are not.

How is the receiving device to know the bounds of each data unit?

Serial data stream: ---01100101110110001011 1--

Where are the byte boundaries?

Clearly our synchronous protocol must define the length of individual data units — and must provide the receiving device with some way to synchronize on data unit boundaries. The SYNC character is used for this purpose. **Every synchronous data stream begins with either one or two SYNC characters:**

SYNC 1 SYNC 2

0110100101101001 1011...

First character of data stream, and data unit boundary.

The data unit in a synchronous, serial data stream usually consists of data bits without parity; but a parity bit may be present. Here is an example of a 9-bit data unit; eight data bits and a parity bit:

x x x x x x x x P

Parity Bit

Eight Data Bits

Either odd or even parity may be specified.

Eight data bits need not always be transmitted; options allow 5, 6, 7 or 8 of the data bits to be meaningful. If less than eight data bits are meaningful, then the balance of high order bits are ignored. Here is an example of a data unit in which only 6 data bits are significant:

x x x x x x P

Parity Bit (if present)

Six Data Bits

Two high order bits are ignored and not transmitted

SERIAL SYNCHRONOUS HUNT MODE

While waiting for synchronous, serial data to start arriving, a receiving device will enter a "hunt" mode, during which it continuously scans the serial data input trying to match an incoming serial data stream with the standard Sync pattern. If your protocol calls for one Sync character, then the receiving device will start interpreting data as soon as it has matched a single Sync pattern. More frequently, protocol will call for two initial Sync characters, in which case the receiving device will not start decoding data until it has matched two sequential Sync characters.

We have already stated that synchronous data transmission requires the transmitting device to continuously send data. What if the transmitting device does not have data ready to send? Under these circumstances **the transmitting device will pad with Sync characters until the next real character is ready to transmit.** To illustrate this concept, **consider an operator entering data at a keyboard;** the keyboard transmits data using very slow, synchronous serial I/O. The operator has to key enter the message:

Goodb̸morningb̸Mr.b̸Smith.

b̸ represents a space character.

Since the operator will be entering data at variable speed, keyboard serial data transmission will insert a Sync character whenever the operator is slow; thus the message may be transmitted as follows:

Gos̸os̸s̸ds̸b̸s̸s̸mos̸rns̸ingb̸s̸s̸s̸Ms̸rs̸s̸.s̸b̸s̸s̸Sms̸s̸iths̸

s̸ represents Sync characters.

When the receiving device decodes a Sync character in the middle of a message, it will ignore the character, but it will remain in synchronization with the serial data stream, ready to interpret the next character.

Assuming that the message illustrated above is being transmitted in ASCII code, consider this portion of the message: Mr.

The binary parallel and synchronous serial equivalent may be illustrated as follows:

```
                 Sync                Sync      Sync
      Char 1    Char 2    Char 3    Char 3    Char 4    Char 5
---010011011011010011011100101011010011011010011001011101---
      M                   r
```

9 digit characters including an odd parity bit are illustrated.

If ASCII characters only are being transmitted, 8-digit characters, including 7 data digits and a parity digit may be used.

SYNCHRONOUS TELEPHONE PROTOCOL

Now in a real synchronous, serial data stream, when communicating via telephone lines, some additional information must be present.

First of all, we need to consider the needs of two-way communication.

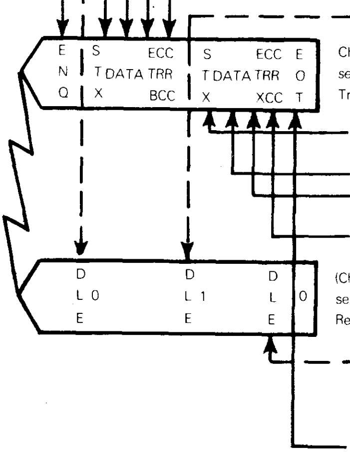
SERIAL DATA HANDSHAKING

BISYNC PROTOCOL

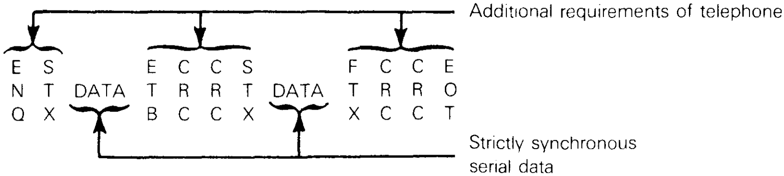
If a serial data communication link is to work, **the transmitting device must be able to transmit commands and receive responses in dialog with the receiving device; this is the only way of insuring that the transmitting device is ready for the transmitted data — or to tell the transmitting device that it must stop transmitting and start receiving.** Following the Sync characters that begin and end any data stream, therefore, there will usually be some well defined control

characters which must be transmitted in both directions. This dialog is referred to as handshaking protocol. For example, Standard IBM 2770 Bisync protocol uses this handshaking sequence:

Transmitter	Receiver
Wishing to transmit, send out an enquiry character to determine whether the receiver is ready to receive.	
	Assuming that it is ready, the receiver sends back a character signifying that it received the enquiry and is ready to receive.
The transmitter now sends a character saying that it is starting text. It then sends a block of data, followed by a character that: (1) indicates the end of the block, and (2) asks the receiver to check the following transmission cyclical redundancy check characters and let the transmitter know if there are any transmission errors.	
	If no errors have been detected, the receiver sends back a different character; this one says, "Yes, I received your block, checked it, found it OK, and now I have received an odd-number of text blocks from you".
The transmitter sends its usual start-of-text character, followed by the text. If this is the last block of data to be transmitted, an end-of-text character is sent, followed by cyclical redundancy check characters.	
	Assuming that the receiver found no errors, it sends back an acknowledging character that says, "Your last block was received OK and now I have received an even number of text blocks."
Upon receiving this response and having nothing more to send, the transmitter simply sends a special end-of-transmission character from the line.	
	Recognizing the special end-of-transmission character, the receiver also disconnects.

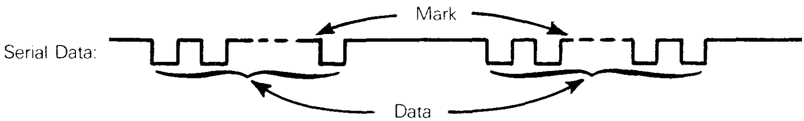


Within the telephone communications sequence, the DATA will be transmitted using synchronous rules previously described:



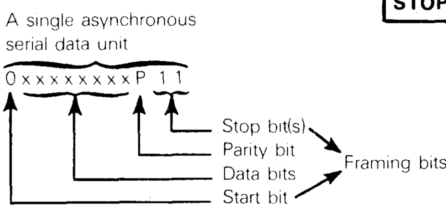
ASYNCHRONOUS SERIAL DATA TRANSFER

When serial data is transferred asynchronously, the transmitting device only transmits a character when it has a character ready to transmit. In between characters a continuous "break" signal, usually a high level, is output:



Every data unit in an asynchronous data stream must carry its own synchronization information. An asynchronous data unit is therefore "framed" by a single Start bit, and one, one and a half, or two Stop bits:

FRAMING
START BIT
STOP BIT

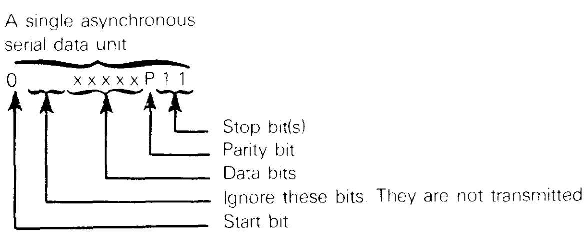


Having a single 0 start bit is universally accepted in the microcomputer world.

There is a similarity between the synchronous data stream's SYNC characters and the framing bits of an asynchronous data stream.

SYNC characters frame a block of synchronous data characters. Start and stop bits frame every data character in an asynchronous data stream.

Of the eight data bits, 5, 6, 7 or 8 may be meaningful, as was the case with synchronous serial data. If less than eight data bits are meaningful, the leftmost, high-order bits are ignored. For example, if your protocol stipulates that there will only be five data bits in each transmitted asynchronous word, then the receiving device will only receive five data bits, and will interpret each received word as follows:



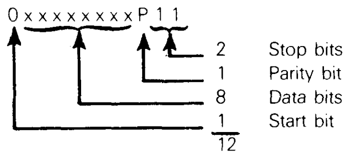
Thus a 9-bit data unit is actually transmitted.

PARITY BIT

The parity bit is always present. Either odd or even parity may be specified.

STOP BITS

1's are always used for stop bits. Most frequently there will be two stop bits; one stop bit is sometimes specified. If you have two stop bits, then every serial 8-bit data word will contain twelve bits:



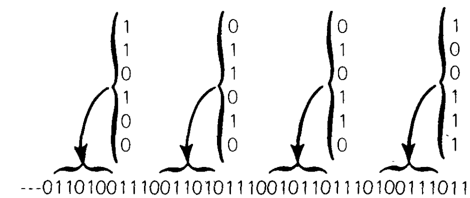
If you have one stop bit, then every serial 8-bit data word will consist of eleven bits.

TELETYPE SERIAL DATA FORMAT

Teletypes use one start bit, seven data bits, a parity bit and two stop bits — for a total of 11 bits per character. Teletypes operate at a standard 10 characters per second, which translates into 110 Baud.

Some transmission protocols specify one and a half stop bits. The stop bit width is one and one half times the normal bit width.

Consider even parity, asynchronous serial data using two stop bits, with 6 data bits in each data unit. This is how a sequence of parallel data will be converted into a serial data stream:



If synchronous serial data communications is occurring over telephone lines, then some form of handshaking protocol, as illustrated for synchronous telephone communications, is going to be required. In fact, there is nothing to prevent the identical handshaking protocol from being used. This protocol is simply a method of transmitting information between two devices via a single telephone line.

FRAMING ERROR

Notice that during asynchronous data transfer the receiving device has an additional means of checking for transmission errors. The first binary digit of every data unit must be a 0 representing the start bit; the last two binary digits of the data unit must both be 1 representing the stop bits. **If the receiving device does not detect appropriate start and stop bits for any data unit in an asynchronous serial data stream, then it will report a framing error.**

A SERIAL I/O COMMUNICATIONS DEVICE

Let us now look at the requirements for a serial I/O interface device.

DUAL IN-LINE PACKAGE SIZE

First of all, how big should the DIP be? We have been using 40-pin DIPs indiscriminately for all of our devices. **Is there some rationale which leads us to a larger or smaller package size, or are we better off simply standardizing on the 40-pin DIP, even if half the pins remain unused?**

The answer is that all other things remaining equal, we would like to use DIPs with as few pins as possible. Bigger DIPs cost more to build and they use up more space on a printed circuit card. Using a 40-pin DIP, where a smaller one would do, can have a snowballing cost effect: Fewer DIPs on a PC card can mean more PC cards. More PC cards can mean a larger backplane, a bigger power supply and a more expensive enclosure.

On the other hand, it makes no economic sense to have a bewildering variety of DIP sizes simply to insure that no DIP ever wastes a pin. For example, you are better off using a standard 40-pin DIP with two unused pins, rather than building an odd-ball 38-pin product.

For our serial I/O communications device we are going to select a 28-pin DIP since this is one of the standard package sizes. We can get away with this smaller number of pins because our serial I/O ports are going to shrink to 1 pin per port.

LOGIC DISTRIBUTION

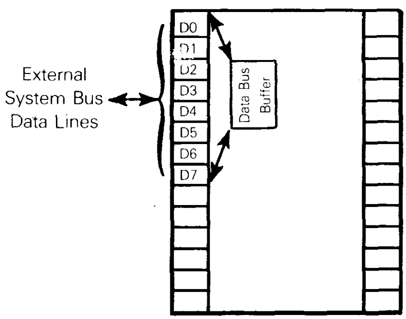
Synchronous and asynchronous serial I/O logic is going to share a single chip. The two sets of logic have enough in common for this to make a lot of sense.

Our serial communications I/O device may be visualized as having three interfaces: One for the microcomputer CPU and one each external asynchronous and synchronous serial I/O. Each interface will, as usual, have data lines and control signals. For the serial I/O interface, control signals can be grouped into general controls and modem controls. General controls apply to any external logic, whereas modem controls meet the specific needs of industry standard modems — which does not prevent you from using modem controls for other external logic if you can.

THE CPU — SERIAL I/O DEVICE INTERFACE

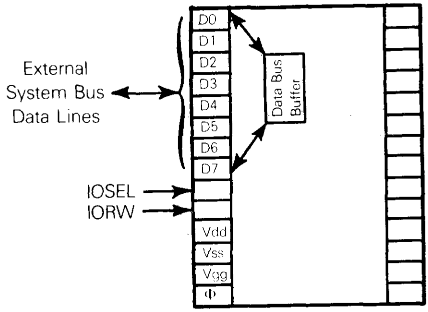
Since the CPU interface is common to synchronous and asynchronous I/O, this is where we will begin.

The serial I/O device is going to communicate in parallel with the CPU via the external system bus data lines. **We must therefore provide 8 data pins, backed up by a data bus buffer:**



Other signals required by the CPU interface are no different from those we included in the parallel I/O interface device. Specifically, these **are IOSEL and IORW.** IOSEL identifies an I/O operation in progress and IORW selects either a 'read to' the CPU or a 'write from' the CPU.

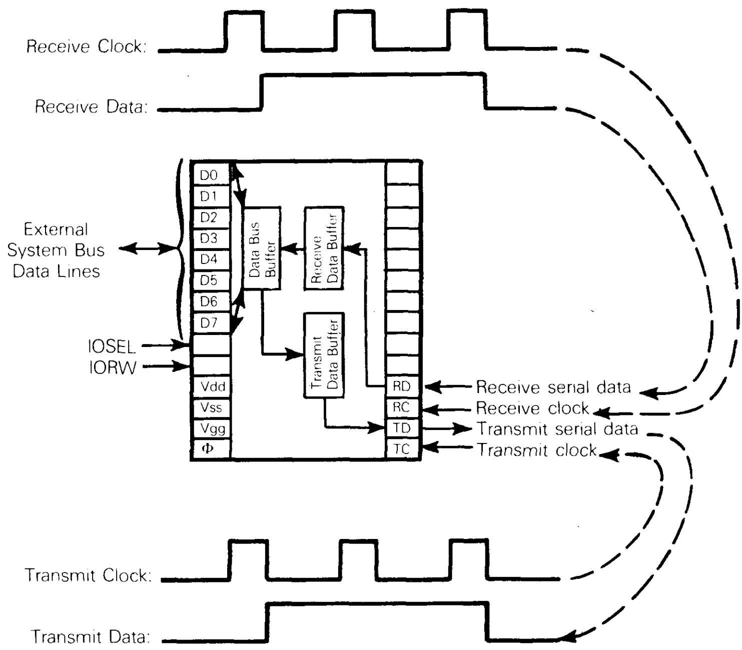
Add clock, power and ground, and our serial I/O interface device looks like this:



THE SERIAL I/O INTERFACE

We are going to use separate pins to transmit and receive serial data. Some devices use a single, bidirectional data pin.

Since we have separate transmit and receive data pins, **we will also need to input separate transmit and receive clock signals.** Both clock signals are input by external logic to control the rate at which data is being transmitted or received:

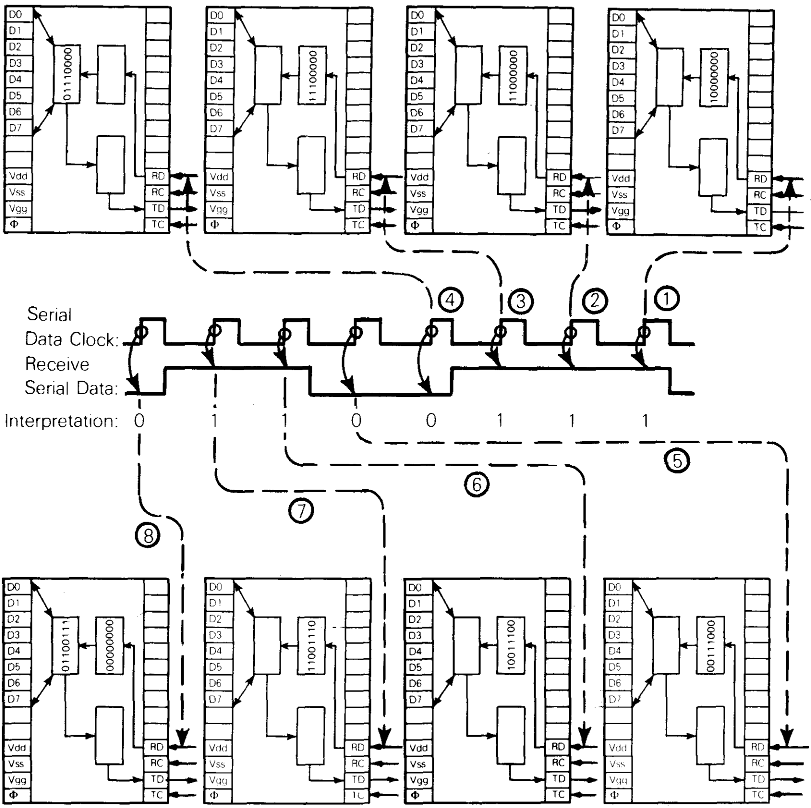


CLOCK SIGNALS

If external logic uses the same transmit and receive clock signal, you can derive both signals from the system clock Φ or from any other clock logic. In any case **these two clock signals are going to control the serial data baud rate.** Baud rate will not be determined by logic internal to the serial I/O interface device and device logic will not generate or output clock signals.

SERIAL DATA INPUT

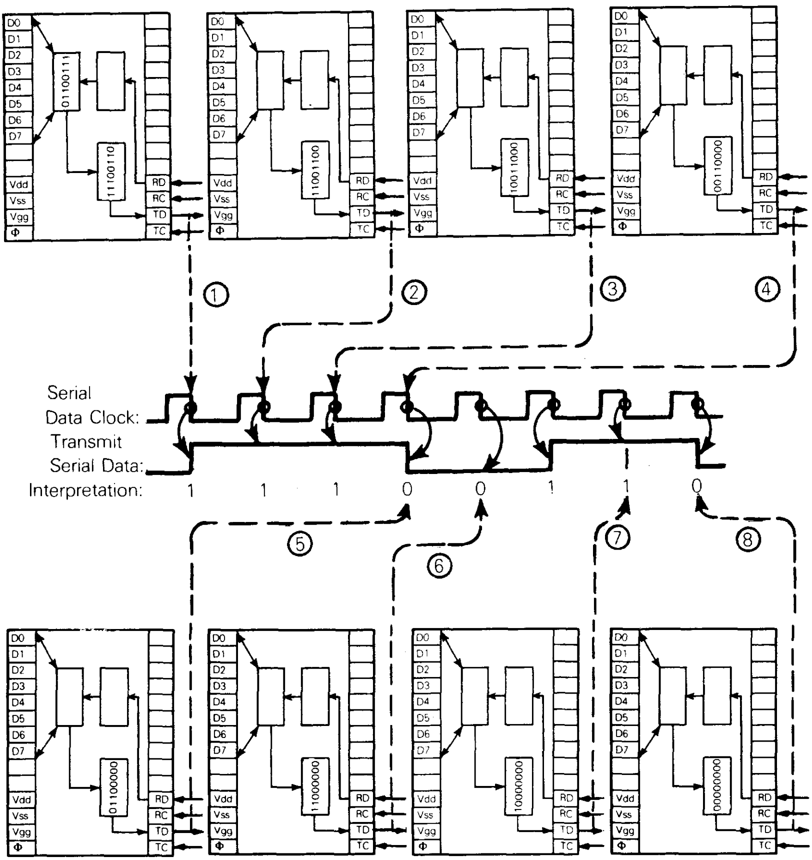
Consider a synchronous data stream where every Receive Clock signal rising edge will strobe the Receive Data signal level, as a binary digit, into the Receive Data buffer. **Whenever the receive data buffer contains eight binary digits, its contents will be transferred to the Data Bus buffer.** Here is an illustration of serial data entry:



The Receive Data buffer is now empty, so the next Receive Data bit will start the loading process all over again.

SERIAL DATA OUTPUT

Every Transmit Clock pulse trailing edge will strobe out a bit from the Transmit Data buffer. **The eight Transmit Data buffer bits will be output in ascending order starting with bit 0.** As soon as bit 7 has been output, the Transmit Data buffer will be considered empty, so the Data Bus buffer contents will be loaded into the Transmit Data buffer, to continue the serial transmit process. Here is an illustration of serial data output:



If asynchronous serial data were being transmitted, the relationship between Data Clock and serial Data signals would change, but that is all. Remember, in an asynchronous data stream you use a x16 or x64 clock and you sample the data on the 8th or 32nd pulse — in the middle of the data bit.

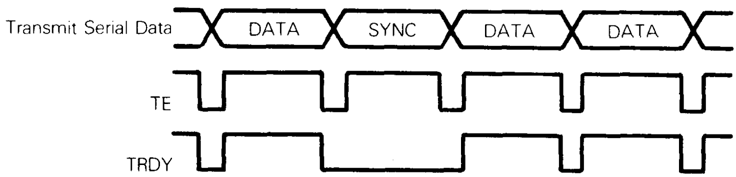
SERIAL I/O CONTROL SIGNALS

The data bus buffer cannot be used simultaneously to receive assembled data bytes and to transmit data bytes for disassembly. **Control logic and control signals which we are now going to describe determine which of the possible operations is occurring at any time.** The serial I/O interface device will simply ignore the clock signal if internal control logic has not been programmed to recognize it. Also, the Receive Data buffer contents will simply be lost if the Data Bus buffer is not ready to receive an assembled byte.

Let us consider the control signals which must be present to support serial data being transmitted and received.

SERIAL TRANSMIT CONTROL SIGNALS

First of all, consider transmit logic; it will need two control signals, one to indicate that the Transmit Data buffer is empty, the other to indicate that the Data buffer is ready to receive another byte of data. We will call these two signals TE and TRDY. The two signals are not identical. For example, when serial data is being output synchronously, TE will be high while a SYNC character is being output; yet TRDY will be low to indicate that the Transmit Data buffer is indeed ready to receive another data byte, even though data is currently being output. Here is the way that TE and TRDY signals will be used:



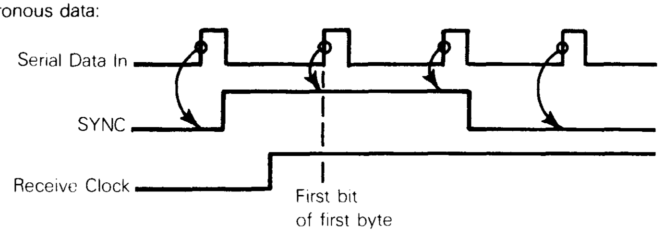
SERIAL RECEIVE CONTROL SIGNALS

Receive logic uses a single Receive Ready signal which we will call RRDY. This signal tells the CPU that a byte of data has been loaded into the Data Bus buffer and can now be read.

Frequently, the RRDY signal will be used to generate an interrupt request. The interrupt can be acknowledged by a very simple instruction sequence that moves the received data byte into an appropriate microcomputer system read-write memory location.

SERIAL RECEIVE SYNCHRONIZATION CONTROL

When synchronous data is being received, remember that the serial I/O interface device logic must detect one or two SYNC characters before acknowledging valid data. External logic must know when the serial I/O device has detected these SYNC characters. **We will therefore add a SYNC control signal, which will be output true as soon as the SYNC characters have been detected.** Some serial I/O devices allow the SYNC control line to be bidirectional. In this case, rather than preceding synchronous data with SYNC characters, **external logic can input the SYNC control signal true;** then the serial I/O device uses this control pulse in order to start receiving synchronous data:



MODEM CONTROL SIGNALS

Only the modem control signals remain to be described. There are these four industry-standard modem control signals:

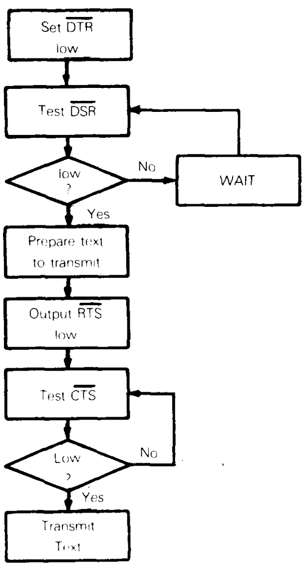
1) $\overline{DSR}$ (Data Set Ready) -- The modem drives this signal low whenever it is ready to receive data. The signal is input high at other times. Any other external logic can use this signal as a master enable/disable. For example, an on/off switch at a unit such as an external video terminal could generate this signal. This allows the microcomputer system to test external logic before attempting to communicate with it.

2) $\overline{DTR}$ (Data Terminal Ready) -- This control signal is the serial I/O device's equivalent of $\overline{DSR}$; it is output by the serial I/O interface device to tell external logic that it is ready to communicate. Under program control you can set this signal high to inhibit all serial I/O operations, or you can set it low to initiate serial I/O operations.

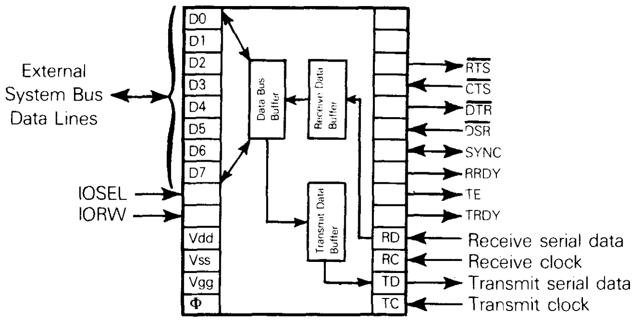
3) $\overline{RTS}$ (Request To Send) -- When the serial I/O device is ready to communicate with a modem or other external logic, $\overline{DSR}$ and $\overline{DTR}$ will both be low. Now the serial I/O device uses the RTS signal to indicate that it is ready to transmit data. Remember that the receiving device may be temporarily busy, even though it has been turned on.

4) $\overline{CTS}$ (Clear To Send) -- In a full duplex data link, $\overline{RTS}$ from the transmitter becomes $\overline{CTS}$ at the receiver; if $\overline{RTS}$ got sent, then clearly there must be a modem at the line end capable of receiving. In a half duplex data link, the modem receiving $\overline{RTS}$ sends back $\overline{CTS}$ two milliseconds later.

The interaction of $\overline{DSR}$, $\overline{DTR}$, $\overline{RTS}$ and $\overline{CTS}$ may be illustrated by the following program flow chart:



This is how our serial interface device now looks:



CONTROLLING THE SERIAL I/O INTERFACE DEVICE

Given the many options available when using the serial I/O interface device, we are going to need a Control Register in order to select options — and in some cases to determine the conditions of control signals being output.

SERIAL I/O MODE

First we must select synchronous or asynchronous I/O; then Table 5-1 identifies the fundamental decisions we must make under program control. **We will refer to Table 5-1 variables as mode parameters,** since they are unlikely to be changed during the course of any serial I/O operation.

FUNCTION	ASYNCHRONOUS	SYNCHRONOUS
Clock frequency	Baud rate x1, x16 or x64	Usually baud rate x1
Data bits per byte	5, 6, 7, or 8	5, 6, 7, or 8
Parity	Odd, even or none	Odd, even or none
Stop bits	1, $1^1/_2$ or 2	Does not apply
Sync characters	Does not apply	1, 2 or external Sync

Table 5-1. Serial I/O Mode Parameters

ISOSYNCHRONOUS SERIAL I/O

Asynchronous I/O using a x1 clock is sometimes called isosynchronous I/O; it is equivalent to transmitting data using asynchronous character format (including framing bits) in an otherwise synchronous data stream.

SERIAL I/O COMMANDS

Within any selected set of mode parameters, the Serial I/O Interface device must still receive commands. Commands must identify the direction of serial data flow (transmit or receive), or terminate current operations, allowing the mode to be modified. Commands must also set the condition of the $\overline{DTR}$ and $\overline{RTS}$ control signals, and respond to any error conditions.

SERIAL I/O ERROR CONDITIONS

What are the error conditions that commands must take care of, and how is the microcomputer system to detect them? **We will provide the Serial I/O Interface device with an 8-bit Status register.** Having 8-bits, we can read a combination of eight input signal statuses and error conditions. **The input signals whose level we must be able to read are:**

SERIAL I/O INPUT CONTROL SIGNALS

1) $\overline{DSR}$ — Data set ready.
2) $\overline{CTS}$ — Clear To Send. This signal is sometimes left out of the Status register; Serial I/O interface device logic must then automatically wait for $\overline{CTS}$ true before initiating a serial data transfer.
3) SYNC — External synchronization.
4) TE — Transmit buffer empty.
5) TRDY — Transmit buffer ready to receive data from the CPU.
6) RDY — Receive buffer ready to send data to the CPU. This signal may be connected to interrupt logic and left out of the Status register.

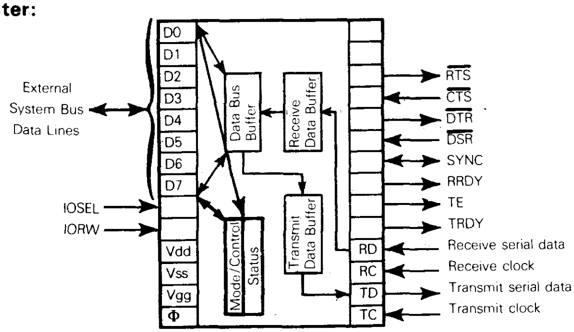
These are the error conditions that may be reported:

1) Parity error. The wrong parity was detected in a serial data unit.
2) Framing error. In asynchronous mode, start and/or stop bits were not correctly detected.
3) Overrun error. The Receive Data buffer transmitted a byte of data to the Data Bus buffer, which was not ready to receive data. The data has been lost.

Normally an error condition does not cause a Serial I/O Interface device to abort operations. The error is reported in the Status register and operations continue undaunted. **Using commands, we will react to an error condition in one of these ways:**

1) In Synchronous mode, send a NAK (no acknowledge) character back to the transmitting source.
2) In Asynchronous mode, abort operations and set TD to its "break" signal level (usually high).
3) Execute any other error recovery program.
4) Reset any error bits in the Status register of the Serial I/O Interface device.

To our Serial I/O Interface device we must now add a Mode/Control register and a Status register:



ADDRESSING THE SERIAL I/O INTERFACE DEVICE

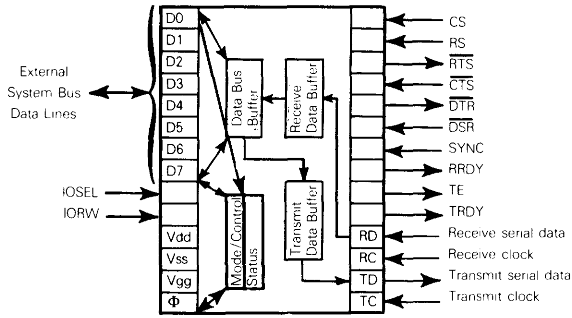
The only aspect of the serial I/O Interface device that we have not covered is how the device is going to be selected, and how its buffers and registers are addressed.

So far as the CPU logic is concerned, the device consists of the Data Bus buffer, the Control register and a Status register.

The Receive Data and Transmit Data buffers lie passively in the path of received and transmitted data, respectively; they communicate with the Data Bus buffer, therefore do not need additional direct access.

In reality the Control and Status registers can be looked on as a single addressable unit, since you can only write into a Control register and you can only read from a Status register.

Thus, we only need two pins in order to access a Serial I/O Interface device — which is just as well, because we only have two pins left. One pin (CS) will constitute a chip select, while the other pin (RS) selects either the Data Bus buffer or the Control/Status register:



Since we only have two pins to address the Serial I/O Interface device, some external logic will be required to appropriately decode address lines, external Data bus lines or control lines in order to create the CS and RS select signals. Use of such external select logic is the rule rather than the exception within the microcomputer industry. In reality, I/O interface devices will not have 8 address lines, as we indicated earlier in this chapter for the parallel I/O interface device.

In Figure 5-18 we can now illustrate one way in which a serial I/O interface device may be integrated into our hypothetical microcomputer system.

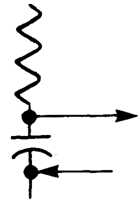
REAL-TIME LOGIC

The concept of real-time logic is one that is easy enough to understand. The most obvious example is the maintenance of real time of day by a microcomputer system that is driving an employee badge reader, and must therefore record the exact time when each employee enters and leaves his place of work.

At the other end of the time scale, a microcomputer may drive an instrument that measures the rotation speed of a drum, fan or propeller, and reports rotation speed in thousands of revolutions per minute.

A microcomputer will have no difficulty keeping track of real time of day since the whole microcomputer system is driven by a clock signal. All that is needed is to add some logic which counts clock signals and generates an interrupt request after a specified number of signal counts. For example, if a clock signal has a period of 500 nanoseconds, then a one millisecond time interval could be clocked by generating an interrupt evey 2000 clock periods. Of course, if the microcomputer clock signal is going to be used to measure real time of day, then a very precise time interval is required. When used in an application that is not time-sensitive, a

microcomputer may have a very inexpensive crystal generating its clock signal. Indeed, in some cases a resistor-capacitor network may be used in the place of a crystal:



Some of the microcomputers described in Volume II provide a programmable timer as an integral part of device logic. When using other microcomputers, all that is needed is some form of pulse count logic external to the microcomputer system, generating an interrupt request after a fixed number of clock periods have been counted.

LOGIC DISTRIBUTION AMONG MICROCOMPUTER DEVICES

The individual devices that we have described in this chapter fairly represent the bulk of microcomputers being sold today. However, there is no reason why logic should be distributed among different devices as described; and if you look at the real devices described in Volume II, you will see that there is indeed wide variation between the logic that one microcomputer manufacturer will implement on single chips as compared to another.

The tradeoff is power versus number of devices. At any given time semiconductor manufacturers can implement a certain number of gates worth of logic on one chip. What this logic should be is the chip designer's business. Nothing but tradition says that a Central Processing Unit and memory must be implemented on separate chips. There is no economic or scientific reason defining chip/logic relationships.

What happens, in reality, is that a chip designer begins by designing a Central Processing Unit. Immediately the chip designer is faced with an important tradeoff; presuming that technology has advanced to the point where he can now put 30% more logic on his chip than the last time around, what is this extra 30% to be? Should the new CPU have a more powerful instruction set with a lot of minicomputer-like addressing modes, or should the instruction set remain the same — with the extra logic being devoted to read-write memory? Or how about using the extra logic to put some parallel I/O on the CPU chip?

In reality the amount of logic which can be crammed onto a single chip is increasing very rapidly. It is for this reason that we are likely to see an equally rapid evolution in microcomputers, with a diverging trend. **At one end of the spectrum we have already reached the one-chip microcomputer, where a minimum representation of all the different logic devices described in this chapter have been crammed onto a single chip. At the other end, we are seeing microcomputers that exactly reproduce minicomputers.**

We, therefore, conclude this chapter with a caution: The segregation of logic on different devices, as described in this chapter, is nothing more than a very approximate guideline; and as time goes by, you will see more microcomputers merging logic onto a very few, or even one device.

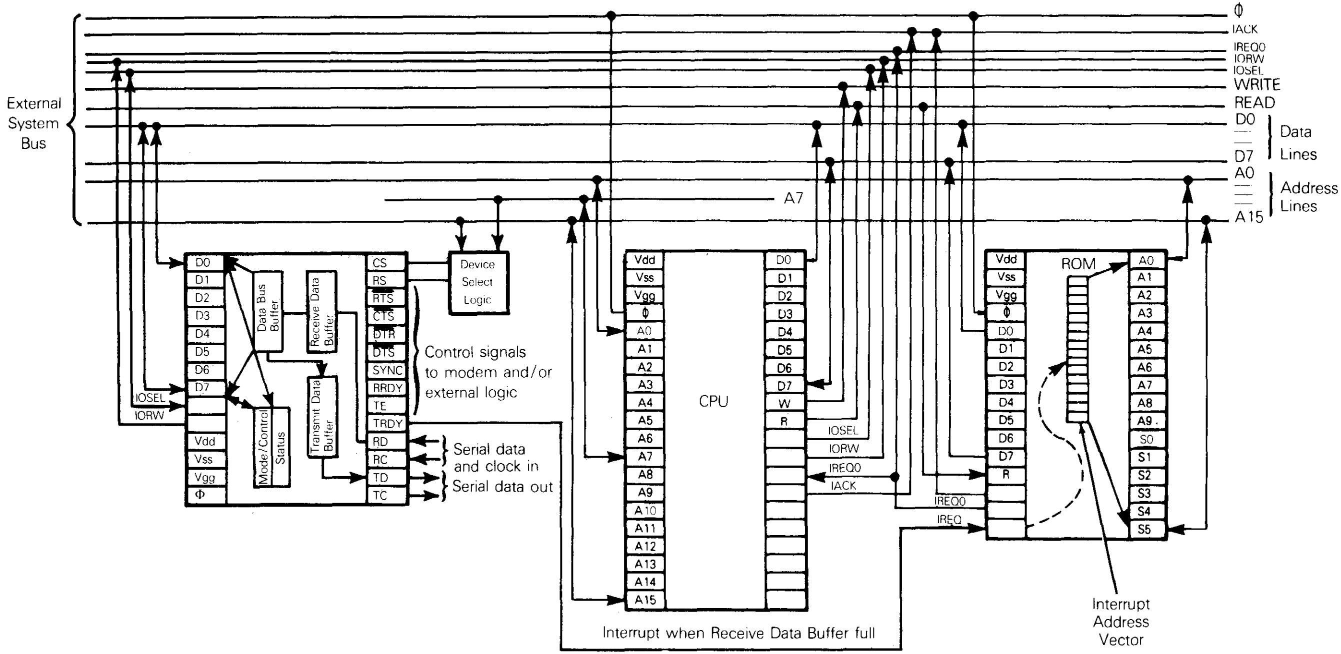


Figure 5-18. Using Serial I/O With Interrupt To Send Received Data To The CPU

Chapter 6
PROGRAMMING MICROCOMPUTERS

Instructions are used to specify any logic sequence that may occur within a microcomputer system. For example, an instruction may complement the contents of the CPU's Accumulator register — or move data from the Accumulator to a memory word — or output data via an I/O port.

To use a microcomputer, therefore, you must first select the devices that will give you sufficient logic capability; then you must sequence the logic to meet your needs, by creating a sequence of instructions which, taken together, select chip logic capabilities that satisfy the needs of your application. The *instruction sequence is a program*, and programming is the creation of the instruction sequences.

THE CONCEPT OF A PROGRAMMING LANGUAGE

The concept of a microcomputer program was introduced in Chapter 3, where a five-instruction, binary addition program was described.

This chapter discusses the types of instructions which a real microcomputer system will need, and how programs are really written. In fact, a discussion of how programs are written must precede the discussion of instruction types, since we are going to use programming terminology in order to describe instructions.

There is nothing to prevent you from creating a computer program as a sequence of binary instruction codes, just as they will appear in memory, or in the Instruction register. The addition program described in Chapter 4 can be written out in binary or hexadecimal digits as follows:

Program As A Binary Matrix	Hexadecimal Version of Program
10011100	9C
00001010	0A
00110000	30
01000000	40
10011100	9C
00001010	0A
00110001	31
10000000	80
01100000	60

Were you to generate your microcomputer program directly as a sequence of binary digits, the chances of misplacing a 0 or a 1 are very high; and the chances of spotting the error are low. This is unfortunate since it is not enough for a program to be 99.99% accurate. Unless it is absolutely accurate, there is always the lurking possibility that the error will manifest itself at an inopportune moment, with disastrous consequences. It is this inherent necessity for perfection that causes programmers to grasp at any device which makes errors harder to create and easier to spot.

As compared to creating a program as a sequence of binary digits, the first and *most obvious improvement would be to code the program using hexadecimal*

digits, then find some automatic way of converting the hexadecimal digits to their binary equivalent.

Writing the program in hexadecimal digits makes it harder to generate errors, because there is one hexadecimal digit for every four binary digits. On the theory that every digit offers an equal probability of being written down wrong, programming in hexadecimal digits is likely to generate one quarter the number of errors, because there are one quarter the number of digits.

Programming in hexadecimal digits also makes errors easier to spot, since detecting a misplaced hexadecimal digit, while not the simplest thing in the world, surely beats spotting a wrong 1 or 0 in a mesmerizing binary pattern. The binary and hexadecimal programs are reproduced below, each having one error. See how long it takes you to find the errors:

Program As A Binary Matrix	Hexadecimal Version of Program
10011100	9C
00001010	0A
00110000	30
01000000	40
10011100	9C
00001010	A0
00011001	31
10000000	80
01100000	60

In the end, however, the program must be converted into a binary sequence, because that is how it is going to be stored in memory — and that is how each instruction must be represented in the Instruction register.

SOURCE PROGRAMS

A teletype, or any other terminal with the appropriate keyboard, generates ASCII character codes in response to keystrokes; therefore, let us assume that a program written in hexadecimal digits will initially be generated as a sequence of ASCII character codes.

Hexadecimal digits are represented by the digits 0 through 9 plus the letters A through F. The ASCII codes for these digits are extracted from Appendix A:

Hexadecimal Digit	ASCII Code
0	00110000
1	00110001
2	00110010
3	00110011
4	00110100
5	00110101
6	00110110
7	00110111
8	00111000
9	00111001
A	01000001
B	01000010
C	01000011
D	01000100
E	01000101
F	01000110

Suppose you write the binary addition program on a piece of paper, using hexadecimal digits, as illustrated in Figure 6-1. This is a source program.

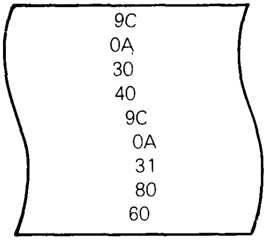
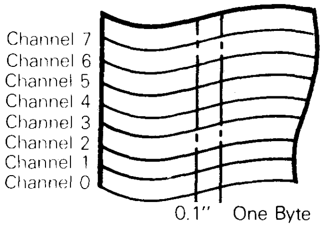


Figure 6-1. A Source Program Written On Paper

This source program must be converted into a form that can be loaded into memory and executed. One way of doing this uses paper tape.

PAPER TAPE

A paper tape has eight "channels", representing the eight binary digits of a byte. A hole punched in any channel represents a 1, while the absence of a hole represents a 0. Ten bytes are represented by one inch of paper tape. In other words, every 0.1" of paper tape represents one byte, as follows:

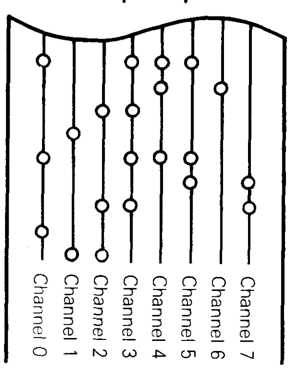


Usually a line of sprocket holes appears between Channels 2 and 3; the sprocket holes are used to a toothed wheel to advance the paper tape.

OBJECT PROGRAMS

Our goal is to convert the source program, illustrated in Figure 6-1, into a paper tape, as illustrated in Figure 6-2. The paper tape in Figure 6-2 is an exact representation of the binary instruction codes that will be stored in memory; 1 digits are represented by holes, and 0 digits are represented by a lack of holes. **The program illustrated in Figure 6-2 is called an Object program.**

Paper Tape



Binary Equivalent

10011100
00001010
00110000
01000000
10011100
00001010
00110001
10000000
01100000

Figure 6-2. An Object Program On Paper Tape

CREATING OBJECT PROGRAMS

Converting the source program of Figure 6-1 into the object program of Figure 6-2 is a two-step procedure.

First the hexadecimal digits illustrated in Figure 6-1 are entered at a keyboard. We will assume it is a teletype keyboard. Each digit becomes an ASCII code on paper tape, as illustrated in Figure 6-3.

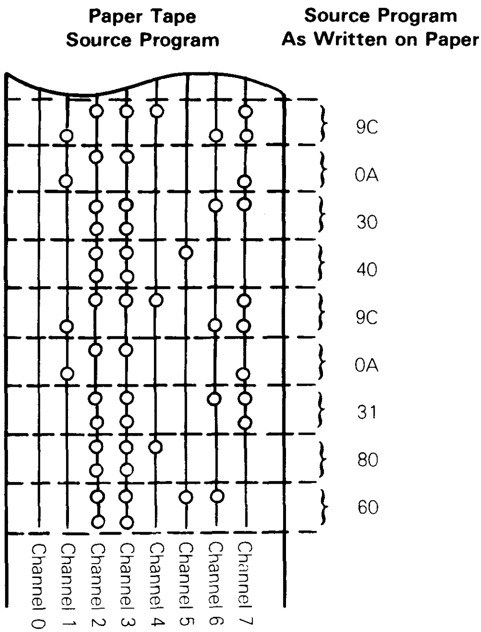


Figure 6-3. A Paper Tape Source Program

EDITORS

You could create the paper tape illustrated in Figure 6-3 by simply turning a teletype punch on, then depressing appropriate keys at the keyboard.

You could get a little more fancy by attaching the teletype to a computer, which executes a program to read keyboard data and punch paper tape. This program is called an EDITOR.

Using an Editor program to create source programs is a good idea. For example, the Editor program could be written to ignore any key that is not a valid hexadecimal digit (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F). Since a teletype can read as well as punch paper tapes, the Editor can read old source program paper tapes, let you make corrections, then punch out the corrected version of the source program. This saves the time you would otherwise spend rekeying the error-free portions of the source program.

Having used an Editor to create a source program on paper tape, as illustrated in Figure 6-3, you will execute another program which automatically reads the source program and creates an object program equivalent; for the moment we will refer to this as a CONVERTER program.

With reference to Figures 6-2 and 6-3, the logic of the Converter program is quite simple and is as follows:

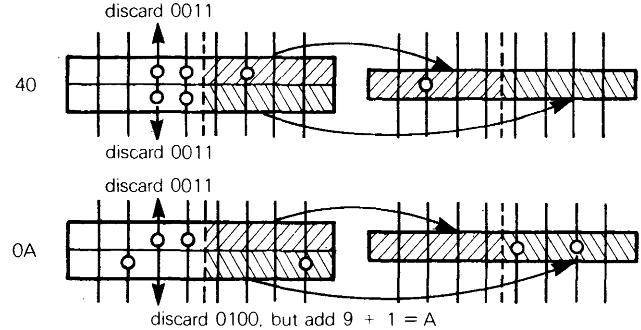
1) Combine the rightmost (low order) four bits of every pair of source program bytes into one object program byte.

2) If Channels 0 through 3 of the source program contain 0011, discard these four bits and use Channels 4 through 7 as is.

3) If Channels 0 through 3 of the source program contain 0100, discard these four bits and use 9 plus the contents of the four bits in Channels 4 through 7.

These three logic steps may be illustrated as follows:

Source Program As Written On Paper	Source Program From Figure 6-3	Object Program In Figure 6-2



The object program paper tape, as created by the Converter program, can be loaded directly into memory to be executed. Chapter 20 of "An Introduction To Microcomputers: Volume II — Some Real Products" describes how this is done for microcomputer systems.

PROGRAM STORAGE MEDIA

You do not have to use paper tape as the medium for creating source and object programs; in fact, only the simplest microcomputer systems will use paper tape. **Usually a magnetic medium, such as a disk unit, is used to store source and object programs (or any other data).**

ASSEMBLY LANGUAGE

Why are hexadecimal digits more efficient than binary digits as a programming medium? Because hexadecimal digits make the programmer's job easier, leaving the hard job to the computer.

Easing the programmer's job — by making errors harder to introduce and easier to spot — is a significant efficiency.

Making the computer convert a hexadecimal source program into a binary object program — by executing a Converter program — is an insignificant penalty, because the Converter program will execute in seconds, or (at most) a few minutes.

Let us take this line of reasoning a step further. **Instead of programming in hexadecimal digits, we will use a programming language which is even simpler for the human programmer to comprehend.**

The programming language source will be very unlike a binary digit object program, so the Converter program, which converts the programming language source program into a binary object program, becomes more complex; but that remains an insignificant penalty.

What a programming language tries to do is eliminate syntactical programming errors — the misplaced digit, the wrong instruction code — leaving only logic errors, specific to the application, as the programmer's responsibility.

Assembly language is the first step into programming forms more easily understood by the human programmer.

ASSEMBLY LANGUAGE SYNTAX

The assembly language of any mini or microcomputer consists of a set of instructions, each of which occupies one line of the source program. Each line may be divided into four parts, or fields, as follows:

Label	Mnemonic	Operand	Comment
	LIM	DC0,ADDR1	;LOAD THE SOURCE ADDRESS INTO DC
HERE	LMA		;LOAD DATA WORD INTO ACCUMULATOR
	AIA	H'0F'	;MASK OFF HIGH ORDER FOUR BITS
	BZ	OUT	;JUMP OUT IF RESULT IS 0
	SRA		;STORE MASKED DATA
	INC	DC0	;INCREMENT THE DATA COUNTER
	JMP	HERE	;RETURN FOR NEXT BYTE
OUT			;NEXT INSTRUCTION

MNEMONIC FIELD

Every source program instruction represents one object program instruction.

Consider first the mnemonic field, which may be highlighted as follows:

Label	Mnemonic	Operand	Comment
	LIM	DC0,ADDR1	;LOAD THE SOURCE ADDRESS INTO DC
HERE	LMA		;LOAD DATA WORD INTO ACCUMULATOR
	AIA	H'0F'	;MASK OFF HIGH ORDER FOUR BITS
	BZ	OUT	;JUMP OUT IF RESULT IS 0
	SRA		;STORE MASKED DATA
	INC	DC0	;INCREMENT THE DATA COUNTER
	JMP	HERE	;RETURN FOR NEXT BYTE
OUT			;NEXT INSTRUCTION

The mnemonic field is the most important field in an assembly language instruction, and is the only field which must have something in it. This field contains a group of letters which constitute a code identifying the source program instruction.

ASSEMBLER

The converter program used to convert assembly language source programs into binary object programs is called an ASSEMBLER. The Assembler reads the mnemonic field, as a group of ASCII characters, and substitutes the instruction binary code in order to generate an object program.

Consider the instruction specified by the mnemonic SRA. This instruction performs the same operation as Instruction 5 of the binary addition program described in Chapter 4, where it is shown using the instruction code 60_{16}. The Assembler must therefore have logic which generates the instruction code 60_{16} upon encountering SRA in the mnemonic field of a source program instruction:

Source Program	Binary form of ASCII mnemonic seen by Assembler		Binary form of Object program created by Assembler
S	01010011		
R	01010010	Assembler logic →	00110000
A	01000001		

Note carefully that only the binary instruction codes of a microcomputer, that is, the object code, are sacred and unalterable. The source program mnemonics are arbitrarily selected and can be changed at any time simply by rewriting the Assembler to recognize the new source program mnemonic.

Every microcomputer's programming manual will define instructions using source program mnemonics. The fact that selection of mnemonics is a very arbitrary business is demonstrated by the fact that only in rare cases will two different microcomputers use the same mnemonic to identify instruction codes that do the same thing. In fact, the selection of instruction mnemonics can become a very emotional issue. Many Intel 8080 users, for example, have created their own set of instruction mnemonics and have gone to the trouble of writing their own Assemblers to recognize their new mnemonics.

Consider the use of more than one mnemonic to represent the same instruction.

We have just shown how the mnemonic SRA is converted to the object program instruction code 60_{16}. Another Assembler could be written to convert the mnemonic XYZ to the object program instruction code 60_{16}. A third Assembler could be written to convert either SRA or XYZ to 60_{16}.

The mnemonics used in this chapter have been selected as typical of those used by real microcomputers.

The mnemonics LIM, LMA, AIA, BZ, SRA, INC and JMP come from the hypothetical microcomputer instruction set which is created in Chapter 7.

Next we will discuss the Label field, which is highlighted as follows:

Label	Mnemonic	Operand	Comment
	LIM	DC0,ADDR1	;LOAD THE SOURCE ADDRESS INTO DC
HERE	LMA		;LOAD DATA WORD INTO ACCUMULATOR
	AIA	H'0F'	;MASK OFF HIGH ORDER FOUR BITS
	BZ	OUT	;JUMP OUT IF RESULT IS 0
	SRA		;STORE MASKED DATA
	INC	DC0	;INCREMENT THE DATA COUNTER
	JMP	HERE	;RETURN FOR NEXT BYTE
OUT			;NEXT INSTRUCTION

LABEL FIELD

The label field may or may not have anything in it. If there is anything in the label field, it is a means of addressing the instruction. In other words, you do not identify an instruction by its location in program memory (as we did in Chapter 4), because at the time you are writing the program, you may not know where in memory the instruction will finish up. This being the case, you give the instruction a name, or label.

Refer to the example above. The instruction labeled HERE must be identified, because later on there is an instruction which specifies a change of execution sequence. The instruction:

```
        JMP        HERE        ;RETURN FOR NEXT BYTE
```

specifies that the instruction labeled HERE is the next instruction to be executed. This is a Jump instruction; it may be used to illustrate what a label means by drawing the following picture of program execution sequence:

Label	Mnemonic	Operand	Comment
	LIM	DC0,ADDR1	;LOAD THE SOURCE ADDRESS INTO DC
HERE	LMA		;LOAD DATA WORD INTO ACCUMULATOR
	AIA	H'0F'	;MASK OFF HIGH ORDER FOUR BITS
	BZ	OUT	;JUMP OUT IF RESULT IS 0
	SRA		;STORE MASKED DATA
	INC	DC0	;INCREMENT THE DATA COUNTER
	JMP	HERE	;RETURN FOR NEXT BYTE
OUT			;NEXT INSTRUCTION

The Assembler is going to have to keep track of where in memory instructions will finish up, because the Assembler is going to have to replace every label with an actual memory address.

Suppose the object program form of the above assembly language source program is going to occupy memory words as follows:

Object Program Memory Locations	**Label**	**Mnemonic**	**Operand**	**Comment**
03FF,0400,0401		LIM	DC0,ADDR1	;LOAD THE SOURCE ADDRESS INTO DC
0402	HERE	LMA		;LOAD DATA WORD INTO ACCUMULATOR
0403,0404		AIA	H'0F'	;MASK OFF HIGH ORDER FOUR BITS
0405,0406		BZ	OUT	;JUMP OUT IF RESULT IS 0
0407		SRA		;STORE MASKED DATA
0408		INC	DC0	;INCREMENT THE DATA COUNTER
0409,040A		JMP	HERE	;RETURN FOR NEXT BYTE
040B	OUT			;NEXT INSTRUCTION

The Assembler will assign the value 0402 to HERE, and 040B to OUT.

The binary instruction code for the JMP instruction happens to be BC_{16}. If the label HERE has the value 0402_{16}, then the Assembler will convert the source program instruction:

```
JMP     HERE
```

to the three object program bytes:

```
BC
04
02
```

If you moved the program, so that the object code for:

```
HERE    LMA
```

now occupied a program memory byte with address $0C7A_{16}$, then the Assembler would convert:

```
JMP     HERE
```

to the three object program bytes:

```
BC
0C
7A
```

OPERAND FIELD

The Operand field may be highlighted as follows:

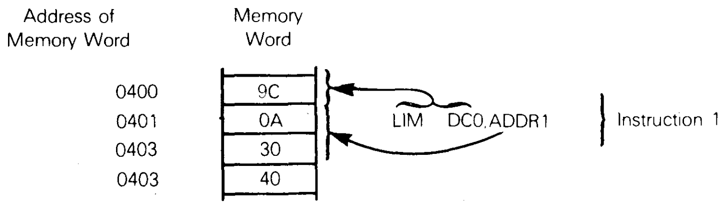
Label	**Mnemonic**	**Operand**	**Comment**
	LIM	DC0,ADDR1	;LOAD THE SOURCE ADDRESS INTO DC
HERE	LMA		;LOAD DATA WORD INTO ACCUMULATOR
	AIA	H'0F'	;MASK OFF HIGH ORDER FOUR BITS
	BZ	OUT	;JUMP OUT IF RESULT IS 0
	SRA		;STORE MASKED DATA
	INC	DC0	;INCREMENT THE DATA COUNTER
	JMP	HERE	;RETURN FOR NEXT BYTE
OUT			;NEXT INSTRUCTION

H'0F' means $0F_{16}$. You cannot use subscripts in a source program, so a reasonable alternative must be selected.

Usually, but not always, the operand field provides information which the Assembler will use to create the second byte or second and third bytes of an instruction that requires more than one byte of object program code. For example, suppose the source program instruction:

```
LIM     DC0,ADDR1
```

is instruction 1 of the binary addition program illustrated in Chapter 3. The Assembler will interpret the source program instruction as follows:

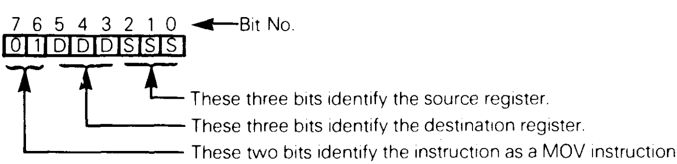


There is no rule which says that the operand must specify or can only specify second and third object program instruction code bytes. The Intel 8080 microcomputer, for example, has seven accumulator-type registers and a single instruction which moves data from one register to another. This source program instruction is written as follows:

```
MOV     D,S
```

where D specifies the destination register.
S specifies the source register.
and D,S constitute the operand field.

But the MOV instruction creates just this one object program byte:



Now look at the contents of the operand fields in the program we are illustrating in this chapter.

DC0,ADDR1 — DC0 identifies the Data Counter into which immediate data must be loaded. ADDR1 is a label representing the address which must be loaded into the Data Counter. The Assembler will convert ADDR1 into a 16-bit binary data value.

H'0F' — specifies the immediate, hexadecimal two-digit value $0F_{16}$. The instruction mnemonic AIA stands for AND Immediate. This combination of mnemonic and operand cause whatever is in the Accumulator to be ANDed with the actual value in the operand field. In this case, since the operand field is $0F_{16}$, it has the effect of setting to zero the high order four bits of the accumulator while leaving the low order four bits of the Accumulator as they are.

OUT and HERE — appearing in operand fields, identify instructions with labels OUT and HERE. The instruction:

```
BZ      OUT      ;JUMP OUT IF RESULT IS ZERO
```

specifies that if the Accumulator contains a zero value following the AND Immediate instruction, then the next instruction to be executed must be the instruction with label OUT, not the SRA instruction which, following sequentially, would normally be executed next. The instruction:

```
JMP     HERE     ;RETURN FOR NEXT BYTE
```

states that the next instruction to be executed must unconditionally be the instruction with the label HERE, not the instruction with label OUT, which, following sequentially, would otherwise have been the next instruction executed.

DCO — specifies the Data Counter whose contents is to be incremented.

COMMENT FIELD

The comment field contains information which makes the program easier to read but has no effect on the binary object program created by the Assembler. In other words, the Assembler ignores the comment field.

FIELD IDENTIFICATION

How is the Assembler going to tell where one field ends and the next begins? Usually space codes are used to separate fields, and the Assembler uses these rules:

1) All characters from the first character on a line up to the first space code constitute the label field.

   If the first character is a space code, then the label field is presumed to be empty.

2) Contiguous space codes are treated as though they were one space code.

3) All characters between the first and second space codes (or contiguous space codes) are interpreted as the mnemonic field.

4) If the mnemonic does not require an operand, the Assembler quits here, assuming everything that follows is comment.

5) If the mnemonic does require an operand, then the Assembler assumes that all characters between the second and third space codes (or contiguous space codes) constitute the operand field.

6) Sometimes comment fields are preceded by a fixed character. We have used the semicolon for this purpose.

Space code field delimiters may be illustrated, according to the above rules, as follows:

Label	**Mnemonic**	**Operand**	**Comment**
	LIM	DCO,ADDR1	;LOAD THE SOURCE ADDRESS INTO DC
HERE	LMA		;LOAD DATA WORD INTO ACCUMULATOR
	AIA	H'0F'	;MASK OFF HIGH ORDER FOUR BITS
	BZ	OUT	;JUMP OUT IF RESULT IS 0
	SRA		;STORE MASKED DATA
	INC	DCO	;INCREMENT THE DATA COUNTER
	JMP	HERE	;RETURN FOR NEXT BYTE
OUT			;NEXT INSTRUCTION

ASSEMBLER DIRECTIVES

An assembly language program, such as the seven-instruction sequence we have been using to illustrate assembly language instruction fields, cannot be assembled as it stands. To give fixed, binary values to labels OUT and HERE, the Assembler must be told where in program memory the object program will eventually reside.

There is a class of instructions, referred to as Assembler Directives, which you will use to provide the Assembler with information that it cannot deduce for itself.

ORIGIN DIRECTIVE

In explaining how the labels OUT and HERE in the operand field would be interpreted by the Assembler, we illustrated the program sequence occupying program memory locations beginning at 03FF. You would specify this origin to the Assembler using an Origin assembler directive as follows:

Label	Mnemonic	Operand	Comment
	ORG	H'03FF'	
	LIM	DC0,ADDR1	;LOAD THE SOURCE ADDRESS INTO DC
HERE	LMA		;LOAD DATA WORD INTO ACCUMULATOR
	AIA	H'0F'	;MASK OFF HIGH ORDER FOUR BITS
	BZ	OUT	;JUMP OUT IF RESULT IS 0
	SRA		;STORE MASKED DATA
	INC	DC0	;INCREMENT THE DATA COUNTER
	JMP	HERE	;RETURN FOR NEXT BYTE
OUT			;NEXT INSTRUCTION

The origin assembler directive generates no object code. Its sole purpose in the program is to tell the Assembler where the object code will be located in program memory, and thus how to calculate the real binary memory addresses that must be substituted for instruction labels.

END DIRECTIVE

Origin is the only assembler directive that is absolutely mandatory. Another assembler directive that is always present, because it makes the job of creating an Assembler easier, is the END directive. This is the last instruction in a program and tells the Assembler that there are no more executable instructions. The END assembler directive may be illustrated as follows:

Label	Mnemonic	Operand	Comment
	ORG	H'03FF'	
	LIM	DC0,ADDR1	;LOAD THE SOURCE ADDRESS INTO DC
HERE	LMA		;LOAD DATA WORD INTO ACCUMULATOR
	AIA	H'0F'	;MASK OFF HIGH ORDER FOUR BITS
	BZ	OUT	;JUMP OUT IF RESULT IS 0
	SRA		;STORE MASKED DATA
	INC	DC0	;INCREMENT THE DATA COUNTER
	JMP	HERE	;RETURN FOR NEXT BYTE
	END		

EQUATE DIRECTIVE

The Equate assembler directive is one more that is universally present because it makes assembly language programming much easier. You use the Equate assembler directive to assign a value to a symbol or a label. Consider the instruction.

AIA	H'0F'	;MASK OFF HIGH FOUR BITS

The operand H'0F' could be replaced by a symbol which is equated to the value $0F_{16}$. This is illustrated as follows:

Label	Mnemonic	Operand	Comment
VALUE	EQU	H'0F'	
	ORG	H'03FF'	
	LIM	DC0,ADDR1	;LOAD THE SOURCE ADDRESS INTO DC
HERE	LMA		;LOAD DATA WORD INTO ACCUMULATOR
	AIA	VALUE	;MASK OFF HIGH ORDER FOUR BITS
	BZ	OUT	;JUMP OUT IF RESULT IS 0
	SRA		;STORE MASKED DATA
	INC	DC0	;INCREMENT THE DATA COUNTER
	JMP	HERE	;RETURN FOR NEXT BYTE
OUT			;NEXT INSTRUCTION

An Equate assembler directive could also be used to assign a value to the address labeled ADDR1. However, you would only do this if ADDR1 did not exist as an instruction label somewhere else within the program.

DEFINE CONSTANT

There are two mnemonics that appear in every assembly language and are neither instructions nor assembler directives. These are the Define Constant and Define Address.

DEFINE ADDRESS

The Define Constant mnemonic is used to specify a single byte of actual data. The Define Address mnemonic is used to specify two bytes of actual data. Here is an example of how these two mnemonics would be used. The instruction sequence:

```
            ORG     H'0700'
            DC      H'3A'
ADDR1       DA      H'27AC'
VALUE       DC      H'0F'
```

would cause the Assembler to directly create the following memory map:

0700	3A
0701	27
0702	AC
0703	0F
0704	
0705	
0706	

MEMORY ADDRESSING

Memory addressing has already been introduced in Chapter 4, where some discussion of the subject was needed in order to define the registers which a CPU will require. We are now going to cover the subject of memory addressing thoroughly as a precursor to defining a microcomputer's instruction set.

MICROCOMPUTER MEMORY ADDRESSING — WHERE IT BEGAN

We will begin our discussion of microcomputer addressing modes by looking at the subject in overall perspective.

The forerunner of all microcomputer instruction sets is the instruction set of the Intel 8008 microcomputer and the Datapoint 2200 minicomputer. These two devices have the same instruction set, but the Datapoint 2200 executes instructions approximately ten times as fast. Mr. Vic Poor (and associates) at Datapoint developed this instruction set for the limited data processing environment of "intelligent terminals." Discrete logic replacement was not what they had in mind. Vic Poor's instruction set was deliberately limited, to accommodate the confines of large scale integration (LSI) technology as it stood in 1969-1970. **The instruction set's memory addressing capabilities were primitive out of necessity, not desire.**

Intel, who initially developed the 8008 microcomputer at Datapoint's request, found a significant market for the product in discrete logic replacement — a market for which the instruction set was never intended.

Subsequent microcomputer instruction sets have evolved as a result of two competing influences:

a) Microcomputer designers incorporated minicomputer features as fast as advances in LSI technology would allow — while there was no definable microcomputer user base.

b) Now that a definable microcomputer user base is beginning to emerge, microcomputer designers are responding directly to users' needs.

Influences (a) and (b), above, do indeed differ. Microcomputer users' needs are not always well suited by minicomputer instruction sets, a fact to which we will continuously return in this chapter.

IMPLIED MEMORY ADDRESSING

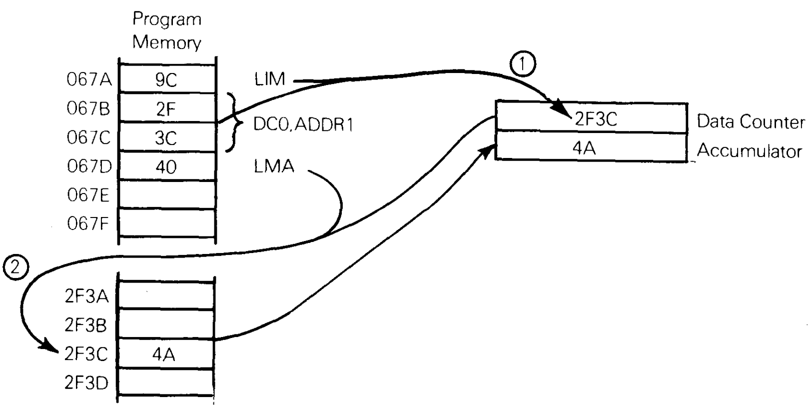
An instruction that uses implied memory addressing specifies the contents of a Data Counter as the memory address.

Implied memory addressing has been described in detail in Chapter 4; therefore, we will simply summarize this addressing mode.

Using the Data Counter to address memory is a two-step process:

- First, the required memory address must be loaded into the Data Counter.
- Next, a single-byte memory reference instruction is executed, where the Data Counter contains the address of the memory location to be referenced.

Consider the first two instructions of the programming example we have been using in this chapter. Execution of the two instructions may be illustrated as follows:



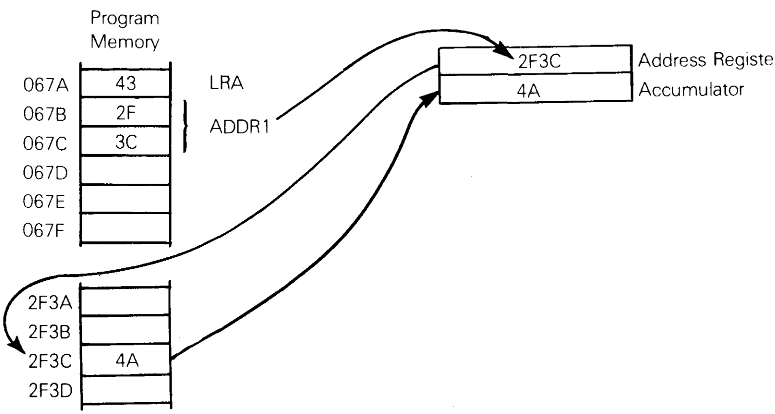
Object code for the first instruction (LIM DC0,ADDR1) occupies three program memory bytes, with addresses 067A, 067B, and 067C. These memory addresses have been arbitrarily selected. Byte 067A contains an 8-bit object program code which represents the LIM DC0,ADDR1 instruction mnemonic. This instruction specifies that the contents of the next two program memory bytes are to be loaded, as a 16-bit value, into Data Counter DC0. Recall that the actual binary code appearing in memory word 067A will vary from microcomputer to microcomputer.

The second instruction, with mnemonic LMA, specifies that the contents of the memory location which is addressed by Data Counter DC0 is to be loaded into the Accumulator. We call this IMPLIED memory addressing, because the memory reference *instruction*, in this case LMA, does not specify a memory address; rather, it stipulates that the memory location whose address is implied by Data Counter DC0 is the memory location to be referenced.

DIRECT MEMORY ADDRESSING

An instruction with direct memory addressing directly specifies the address of the memory location to be referenced.

Simple direct memory addressing has been described in Chapter 4 along with implied memory addressing. In terms of the instruction sequence we are using in this chapter, the LIM and LMA instructions could be combined into one direct memory reference instruction as follows:



An Address Register performs the same function as a Data Counter, but it does so transiently.

A direct memory reference instruction always starts with a memory address being computed and loaded into the Address register. This becomes the address of the memory location to be reference.

Direct addressing is the simplest addressing mode used by minicomputers. Implied memory addressing is a microcomputer phenomenon.

DIRECT VERSUS IMPLIED ADDRESSING

By way of direct comparison, a minicomputer Address register is referred to as a nonprogrammable register. This means that a minicomputer has no instructions that merely load data into the Address register or modify the Address register's contents. The process of changing the contents of the Address register is always one transient step in the course of executing a memory reference instruction.

The Data Counter in a microcomputer is programmable. In fact, every microcomputer will have a number of instructions that simply load data into the Data Counter, or modify the Data Counter contents, but do nothing else.

Some microcomputers provide both implied and direct memory addressing. These microcomputers have one or more Data Counters for implied memory addressing, plus an additional Address register for direct memory addressing.

The first microcomputers used implied memory addressing only, because it was simple to design into the CPU Control Unit; there was no other reason. The penalty incurred by the use of implied memory addressing is that it takes two instructions to do what one direct memory addressing instruction could do. LSI technology has advanced to the point where microcomputer designers could do away with implied memory addressing, but they have not done so. Today most microcomputers include a limited number of instructions with direct memory addressing, but implied memory addressing remains the standard; why? Because some necessary variations of direct memory addressing generate features that are very undesirable when programs are stored in ROM.

VARIATIONS OF DIRECT MEMORY ADDRESSING

We will first consider variations of direct memory addressing as they apply to minicomputers with 12-bit and 16-bit words. This is a good beginning, since direct memory addressing variations evolved as minicomputer memory addressing features.

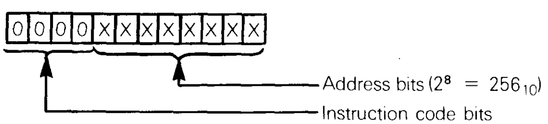
A 16-bit word allows a minicomputer to have 65,536 different instructions in its instruction set. A 12-bit word allows a minicomputer to have 4096 different instructions in its instruction set. These are ridiculously high numbers. Minicomputers therefore separate instruction words into instruction code bits and address bits.

12-BIT WORD DIRECT ADDRESSING

PDP-8

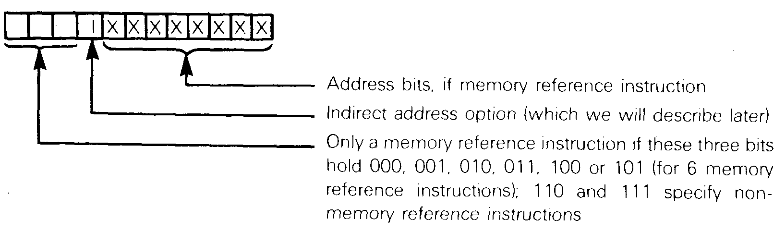
INTERSIL IM6100

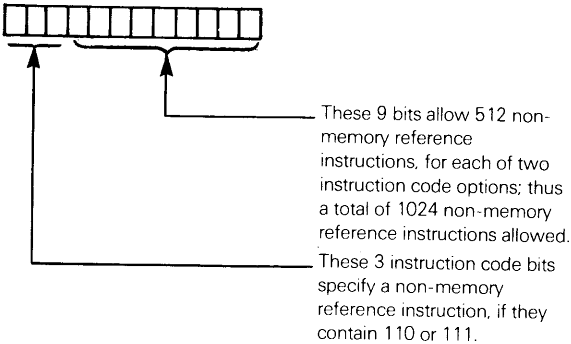
Consider first a minicomputer with a 12-bit word. Digital Equipment Corporation's PDP-8, the world's first popular minicomputer, uses a 12-bit word. The PDP-8 CPU is now manufactured by Intersil on a single chip, called the IM6100. **The 12-bit word may be used as follows:**



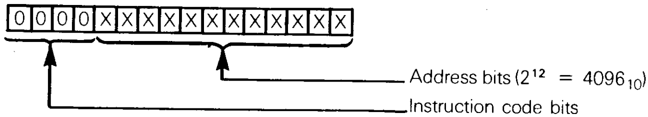
Eight address bits allow this instruction to directly address up to 256 memory words; 256 words constitutes a very small memory, so we will have to seek ways of expanding our memory addressing range, without using more addressing bits.

Will four instruction code bits be enough? Yes indeed. Remember, the above separation of twelve bits into four instruction code bits and eight address bits only applies to memory reference instructions. This is how the PDP-8 and IM6100 interpret memory reference instructions:

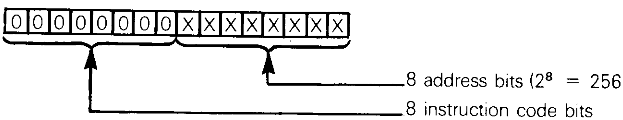




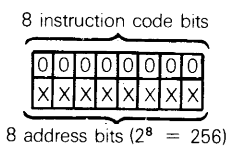
A 16-bit computer could address 4096_{10} memory words with 12 bits of an instruction word:



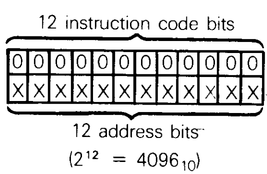
The 16-bit computer could offer more instruction code options, and a smaller addressing range, as follows:



Most minicomputers divide their single word instructions into eight address bits and eight instruction code bits, as illustrated above. An 8-bit microcomputer can easily achieve the same result, using two 8-bit words, as follows:

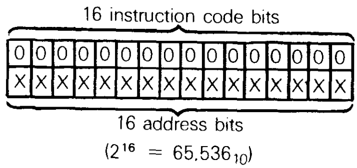


A 12-bit computer could address 4096_{10} words by providing two words per instruction:

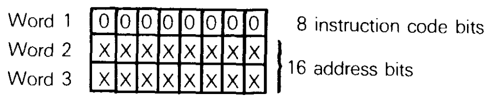


Although two 12-bit words per instruction is very feasible, all PDP-8 instructions are, in fact, one-word instructions.

If a 16-bit computer uses two words per instruction, it can directly address $65{,}536_{10}$ words of memory:

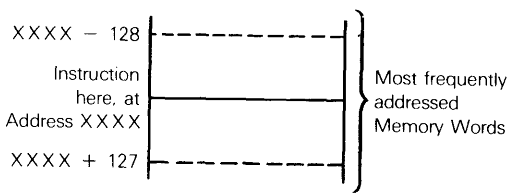


The microcomputer programming examples we have used earlier in this book specify 16-bit memory addresses via three 8-bit words, as follows:



ADDRESS BITS — THE OPTIMUM NUMBER

It is clear that one way or another, instructions can have anywhere from 8 to 16 address bits. What is the optimum number? In minicomputer applications, statistically we find the 80% to 90% of Jump and Branch instructions only need an addressing range ±128 words (addressable with eight address bits):

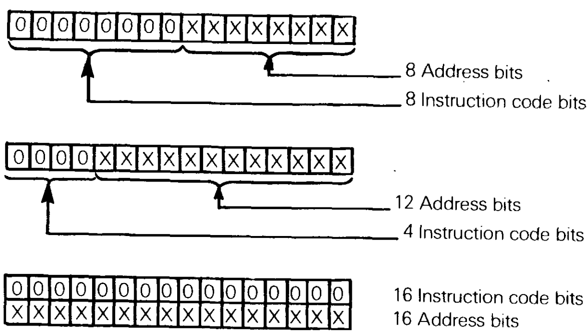


Most minicomputers provide more than one instruction word format, including two or all three of the following:

Format 1 (limited addressing range)

Format 2 (limited addressing range)

Format 3 (extended addressing range)

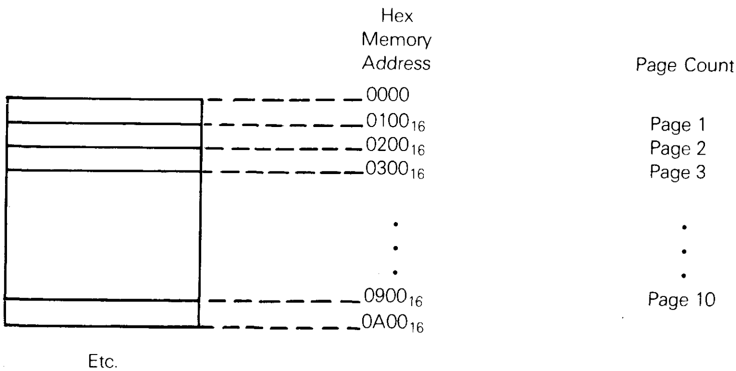


Instructions with a restricted addressing range are referred to as SHORT FORM instruction. LONG FORM instructions have sufficient address bits to directly address any word in memory.

PAGED DIRECT ADDRESSING

All computers provide some instructions with a limited addressing range. If all of a computer's instructions are subject to a limited addressing range, the computer is said to be PAGED.

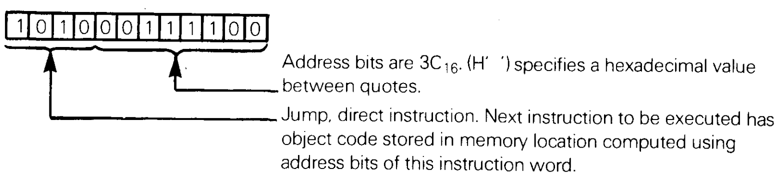
To illustrate paging, consider a 12-bit minicomputer with eight address bits per instruction. Memory is effectively segmented into 256_{10} (100_{16}) word pages as follows:



The eight address bits of an instruction word provide the two low order hexadecimal digits of a four digit memory address; the two high order digits are taken from the Program Counter. Thus the instruction:

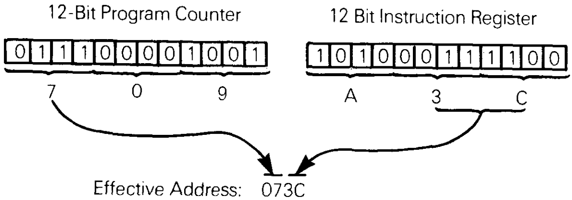
JMP H'3C'

would be coded in one 12-bit word, as follows:



EFFECTIVE MEMORY ADDRESS

Suppose the jump instruction is stored in a memory word with the address 0709_{16}. After the jump instruction has been fetched from memory, the Program Counter will contain the value $070A_{16}$. The effective memory address, therefore, is given by the two high order digits of the Program Counter, plus the two low order digits from the instruction, as follows:



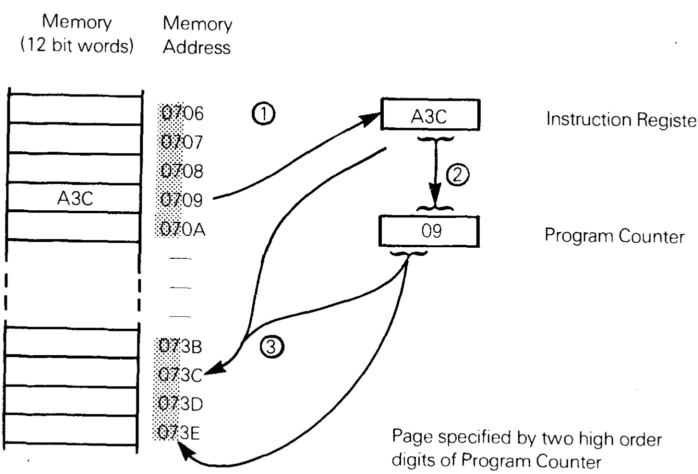
PAGE NUMBER

The high order digits, taken from the Program Counter, are called the page number. The low order digits, supplied by the instruction code, are the address within the page. Combining the two portions of the address yields the effective memory address. The term "effective memory address" applies to any memory address that must be computed in some way using information provided by the instruction.

As illustrated below, the instruction:

JMP H'3C'

will cause the value $073C_{16}$ to be loaded into the Program Counter, so the next instruction will be fetched from memory location $073C_{16}$.



This illustration is described, with reference to the keys ①, ② and ③, as follows:

① The Program Counter addresses memory word 0709_{16}. The contents of this memory word, $A3C_{16}$, is fetched and stored in the Instruction register. The Program Counter is incremented to $070A_{16}$; thus, it addresses the next sequential program memory word.

② The instruction code in the Instruction register is an unconditional jump. The two low order digits of the Instruction register ($3C_{16}$) are moved to the two low order digits of the Program Counter, which now contains $073C_{16}$.

③ The next instruction will be fetched from memory location $073C_{16}$.

Consider another example. The instruction:

```
LMA     H'6C'
```

located on Page $2F_{16}$ would cause the contents of memory location $2F6C_{16}$ to be loaded into the Accumulator. The same instruction on Page $1C_{16}$ would cause the contents of memory location $1C6C_{16}$ to be loaded into the Accumulator.

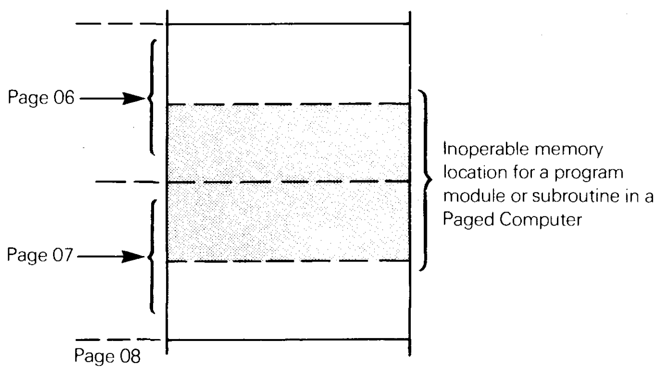
PAGE BOUNDARY ERROR

With many paged computers a devious error occurs at the page boundaries. Recall that the Program Counter is incremented after an instruction has been fetched. Therefore the page number is acquired from the high order digits of the Program Counter, AFTER the Program Counter has been incremented. Suppose the instruction:

```
LMA     H'6C'
```

were located at a memory word with address $2FFF_{16}$. After this instruction has been fetched, the Program Counter will contain 3000_{16}. Now the contents of the memory word with address $306C_{16}$ would be loaded into the Accumulator, instead of the memory word with address $2F6C_{16}$.

The most severe restriction imposed by fixed pages is that an instruction cannot reference any memory word outside the page on which the instruction is located: to read data, to write data or to execute a program branch or jump. Therefore, **programs cannot reside across a page boundary:**



Paging is wasteful of computer memory, because it requires programs to access all data on, or via addresses stored on the program's page. Thus numbers used commonly by programs on many different pages must be stored repeatedly on each page, or else we must add some new flexibility to paged addressing.

Also, when writing program modules and subroutines, it is difficult to contrive that every module exactly fills one page. As a result, a small portion of memory at the end of each page is wasted, since it is too small to accommodate even the smallest subroutine. Thus **a programmer must frequently waste a lot of time juggling the sizes and memory locations of his program modules and subroutines.**

Consider, for example, a program with the following modules:

Program	Size (words)
MAIN	88_{16}
SUB1	22_{16}
SUB2	78_{16}
SUB3	52_{16}
SUB4	38_{16}
SUB5	50_{16}
SUB6	66_{16}

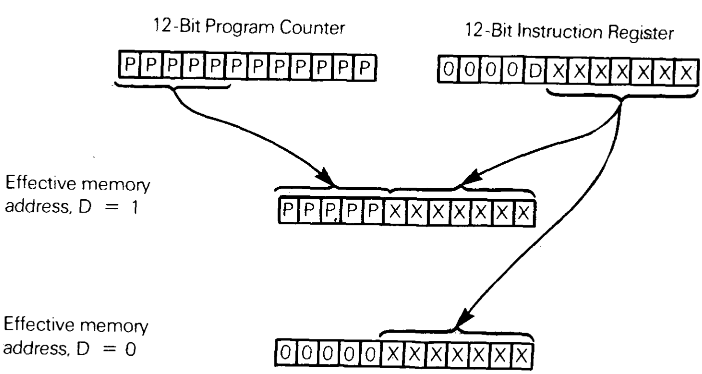
We can map the program into memory as follows:

Program	Memory Location
MAIN	0300_{16} - 0387_{16}
SUB2	0388_{16} - $03FF_{16}$
SUB3	0400_{16} - 0451_{16}
SUB4	0452_{16} - 0489_{16}
SUB5	0490_{16} - $04FF_{16}$
SUB6	0500_{16} - 0565_{16}
SUB1	0566_{16} - 0585_{16}

But beware of an error in subroutine SUB3 that requires you to increase its size (by two instructions, say). Subroutines SUB3, SUB4, and SUB5 no longer fit on one page, and correcting SUB3 will require remapping the whole program in memory.

BASE PAGE

One method of eliminating some of the restrictions imposed by paged addressing is to provide the computer with a base page. This is what the PDP-8 does, so let us look at this specific case. In order to give itself one more option, the PDP-8 uses just seven of the eight address bits to compute addresses within a page; in other words, the PDP-8 page is not 256 words long, it is 128 words long. However, the eighth bit allows you to address either the current page, that is, the page on which the instruction is located, or you can address the base page, that is, one of the first 128 words of memory. This is illustrated as follows:



In the above illustration, symbols are used as follows:

P represents individual binary digits with the Program Counter.

0 represents the instruction code bits of the Instruction register.

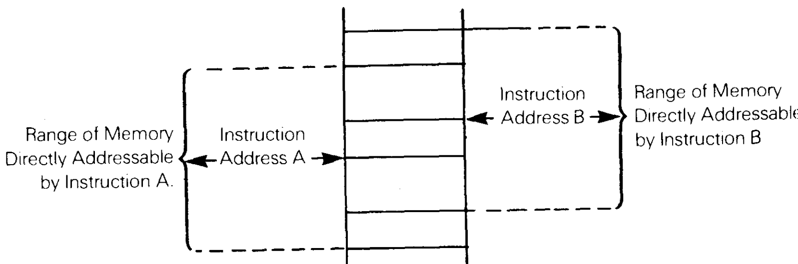
X represents individual address bits of the instruction word.

D is the page select bit. If this bit is 0, then the effective memory address is computed by moving the X bits into the low order bits of an Address register and inserting 0's in the five high order bits of the Address register; in other words, memory locations from 0 to 127 may be addressed. This is referred to as the base page of memory. If the D bit is 1, then the five high order bits of the Address register are taken from the five high order bits of the Program Counter; only memory locations within the 128 word page in which the instruction resides may be referenced.

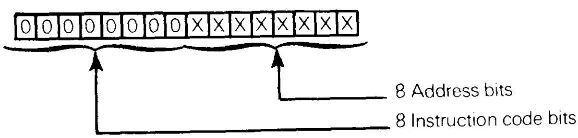
PROGRAM RELATIVE PAGING

A more flexible variation of paging is PROGRAM RELATIVE paging, in which it is assumed that the address bits of an instruction represent a signed binary displacement, which must be added to the Program Counter contents.

A program relative page may be illustrated as follows:



Program relative addressing allows an instruction to address memory in a forward (positive) or backward (negative) direction with a range in each direction of half a page. Consider again an 8-bit address, this time in a 16-bit word:



Assume that the high order address bit is a sign bit; if the instruction is located at memory word $24AE_{16}$, and the eight address bits contain $7A_{16}$, then the effective memory address is given by:

```
  0 0 1 0 0 1 0 0 1 0 1 0 1 1 1 0        24AE
  0 0 0 0 0 0 0 0 0 1 1 1 1 0 1 0       + 7A
  0 0 1 0 0 1 0 1 0 0 1 0 1 0 0 0        2528
```

Sign bit is propagated through high order eight bits.

This example of forward memory addressing may be illustrated using assembly language instructions as follows:

Memory Address	Object Code	Source Program Instruction		
24AD	BC7A		JMP	HERE
—	—		—	
—	—		—	
—	—		—	
2528		HERE	LMA	THERE

In the above illustration, BC represents the JMP instruction code. (The LMA instruction's object code is irrelevant to the discussion at hand, so it is left out.) The Assembler will compute the address bits of the JMP instruction by subtracting the value in the Program Counter, after the JMP instruction has been loaded, from the value associated with the label HERE:

```
Label HERE is equivalent to          2528
PC contents after instruction load = 24AE
                        Difference = 007A
```

If the Assembler computes a value greater than $7F_{16}$, the JMP instruction is illegal, and an error message will be transmitted to the errant programmer by the assembler.

Suppose the two instructions were reversed; we would now have:

Memory Address	Object Code	Source Program Instruction		
24AD		HERE	LMA	THERE
—			—	
—			—	
—			—	
2528	BC84		JMP	HERE

The Assembler will compute address bits of the JMP instruction in the same way, by subtracting the value in the Program Counter, after the JMP instruction has been loaded, from the value associated with label HERE:

```
Label HERE is equivalent to          24AD
PC contents after instruction load = 2529
                        Difference = - 7C
```

−7C is stored in its twos complement form:

```
               7C = 0 1 1 1 1 1 0 0
 ones complement  = 1 0 0 0 0 0 1 1
 twos complement  = 1 0 0 0 0 1 0 0 = 84
```

The effective memory address provided by the JMP instruction is computed as follows:

```
0 0 1 0 0 1 0 1 0 0 1 0 1 0 0 1      2529
1 1 1 1 1 1 1 1 1 0 0 0 0 1 0 0      FF84
0 0 1 0 0 1 0 0 1 0 1 0 1 1 0 1      24AD
```

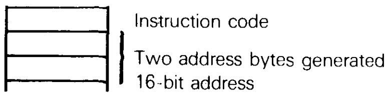
Sign bit is propagated through high order eight bits.

DIRECT MEMORY ADDRESSING IN MICROCOMPUTERS

Variations of direct addressing which are useful in minicomputer applications are not useful, and are frequently not even viable, in microcomputer applications. Let us carefully examine why this is the case.

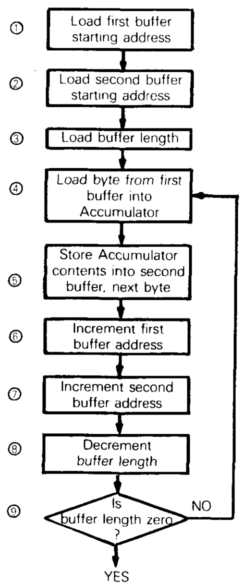
EXTENDED DIRECT ADDRESSING

Consider first a three-byte, direct memory reference instruction, which we have represented as follows:



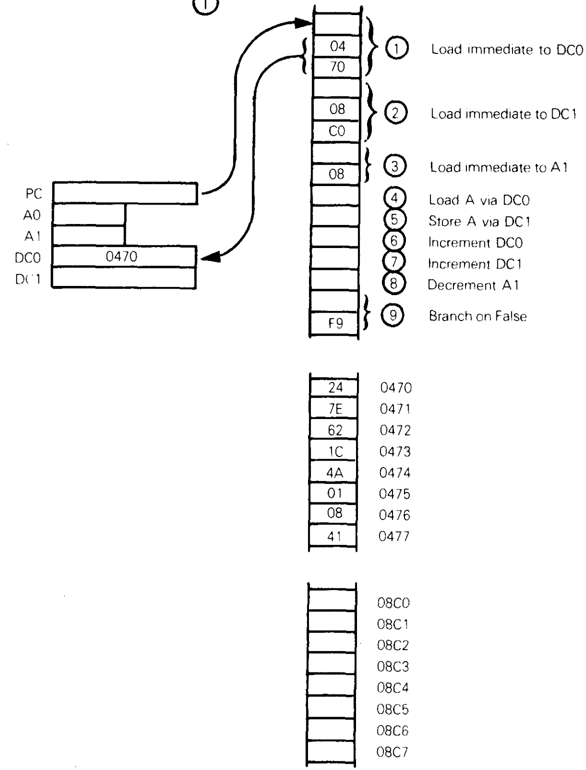
This instruction format will certainly work. The two address bytes allow any memory location to be addressed directly either to store, read or otherwise manipulate data, or to change program execution sequence by executing a jump or branch instruction. **There are, however, two problems associated with the use of three-byte, direct memory reference instructions: first, the two address bytes cannot be changed; second, three-byte instructions are very wasteful of memory.**

Here is an example of an instruction sequence that moves data from one buffer to another buffer:



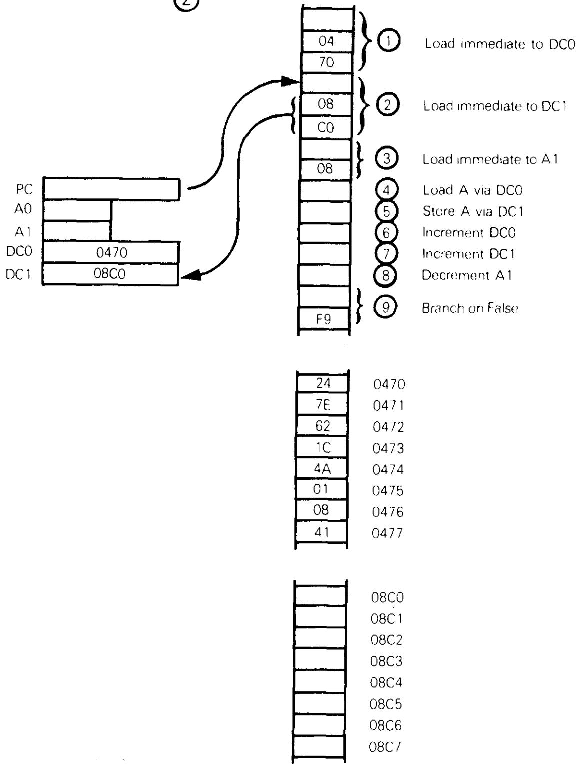
Consider how the above program logic sequence would be executed by a microcomputer that has two Data Counters (DC0 and DC1) and two Accumulators, (A0 and A1). Assume that Buffer 1 begins at memory location 0470_{16}, Buffer 2 begins at memory location $08C0_{16}$, and each buffer is eight bytes long. Illustrating CPU Data register contents, this is what happens:

①

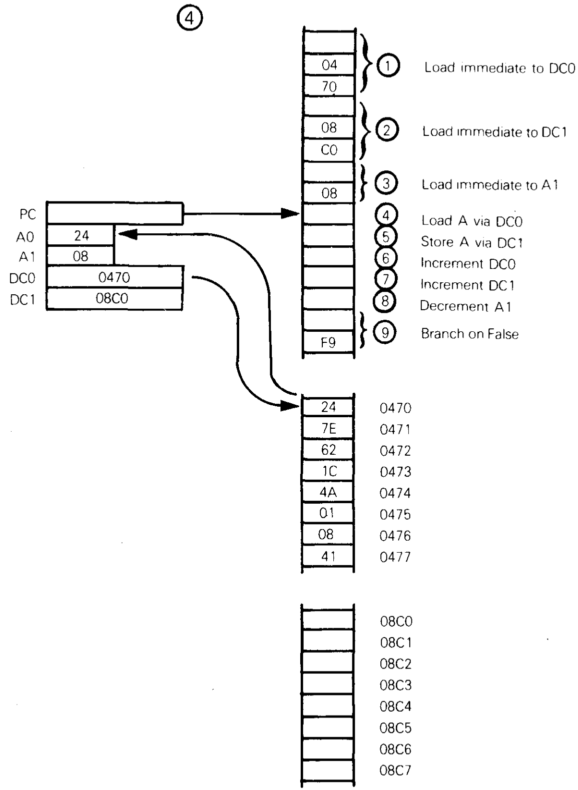


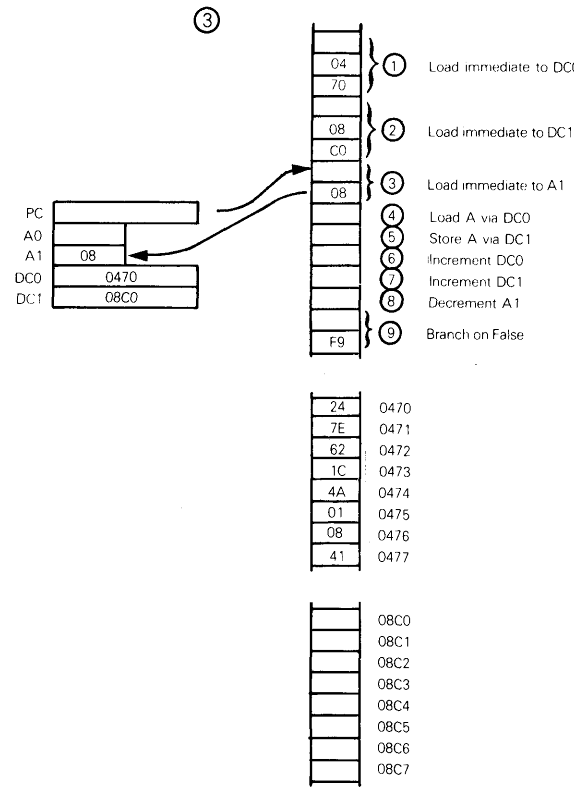
04
70
08
C0
08
F9
Load immediate to DC0
Load immediate to DC1
Load immediate to A1
Load A via DC0
Store A via DC1
Increment DC0
Increment DC1
Decrement A1
Branch on False
PC
A0
A1
DC0
DC1
0470
24
7E
62
1C
4A
01
08
41
0470
0471
0472
0473
0474
0475
0476
0477
08C0
08C1
08C2
08C3
08C4
08C5
08C6
08C7

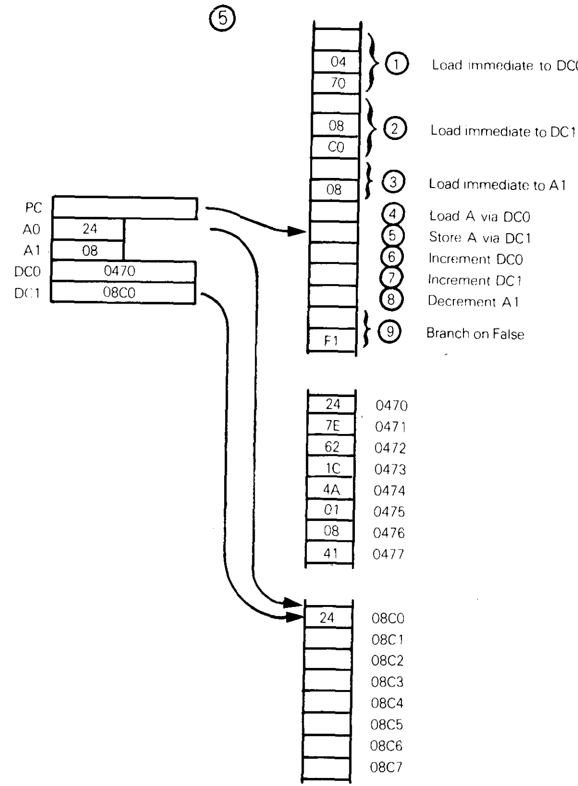
②



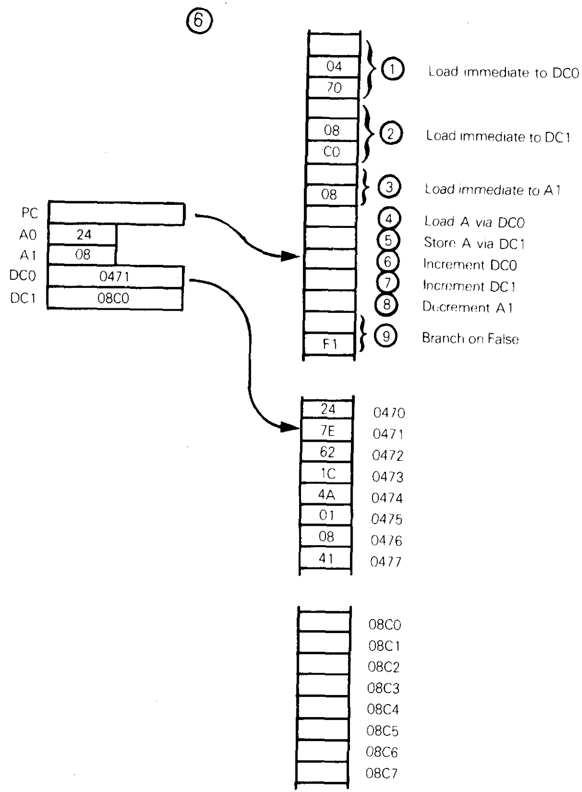
04
70
08
C0
08
F9
Load immediate to DC0
Load immediate to DC1
Load immediate to A1
Load A via DC0
Store A via DC1
Increment DC0
Increment DC1
Decrement A1
Branch on False
PC
A0
A1
DC0
DC1
0470
08C0
24
7E
62
1C
4A
01
08
41
0470
0471
0472
0473
0474
0475
0476
0477
08C0
08C1
08C2
08C3
08C4
08C5
08C6
08C7


④
PC
A0
A1
DC0
DC1
24
08
0470
08C0
04
70
08
C0
08
F9
① Load immediate to DC0
② Load immediate to DC1
③ Load immediate to A1
④ Load A via DC0
⑤ Store A via DC1
⑥ Increment DC0
⑦ Increment DC1
⑧ Decrement A1
⑨ Branch on False
24 0470
7E 0471
62 0472
1C 0473
4A 0474
01 0475
08 0476
41 0477
08C0
08C1
08C2
08C3
08C4
08C5
08C6
08C7


③
PC
A0
A1
DC0
DC1
08
0470
08C0
04
70
08
C0
08
F9
① Load immediate to DC0
② Load immediate to DC1
③ Load immediate to A1
④ Load A via DC0
⑤ Store A via DC1
⑥ Increment DC0
⑦ Increment DC1
⑧ Decrement A1
⑨ Branch on False
24 0470
7E 0471
62 0472
1C 0473
4A 0474
01 0475
08 0476
41 0477
08C0
08C1
08C2
08C3
08C4
08C5
08C6
08C7

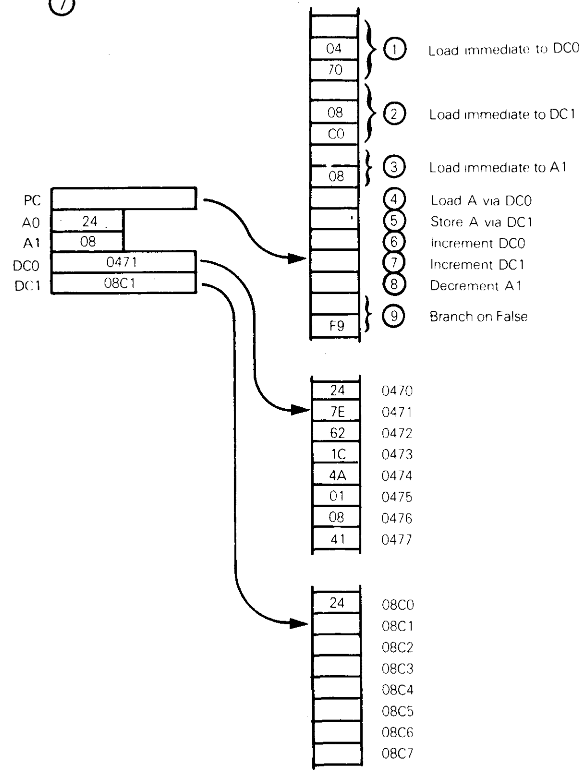


5
PC
A0
A1
DC0
DC1
24
08
0470
08C0
04
70
08
C0
08
F1
1
Load immediate to DC0
2
Load immediate to DC1
3
Load immediate to A1
4
Load A via DC0
5
Store A via DC1
6
Increment DC0
7
Increment DC1
8
Decrement A1
9
Branch on False
24 0470
7E 0471
62 0472
1C 0473
4A 0474
01 0475
08 0476
41 0477
24 08C0
08C1
08C2
08C3
08C4
08C5
08C6
08C7

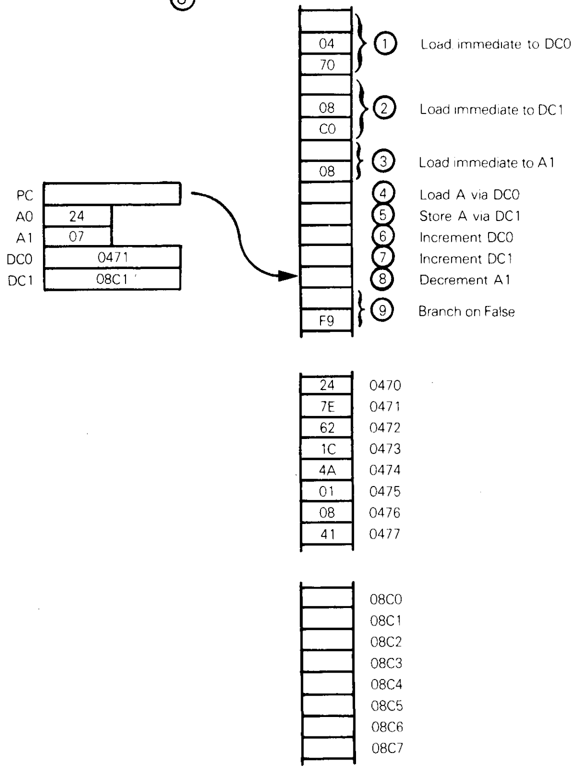


6
PC
A0
A1
DC0
DC1
24
08
0471
08C0
04
70
08
C0
08
F1
1
Load immediate to DC0
2
Load immediate to DC1
3
Load immediate to A1
4
Load A via DC0
5
Store A via DC1
6
Increment DC0
7
Increment DC1
8
Decrement A1
9
Branch on False
24 0470
7E 0471
62 0472
1C 0473
4A 0474
01 0475
08 0476
41 0477
08C0
08C1
08C2
08C3
08C4
08C5
08C6
08C7

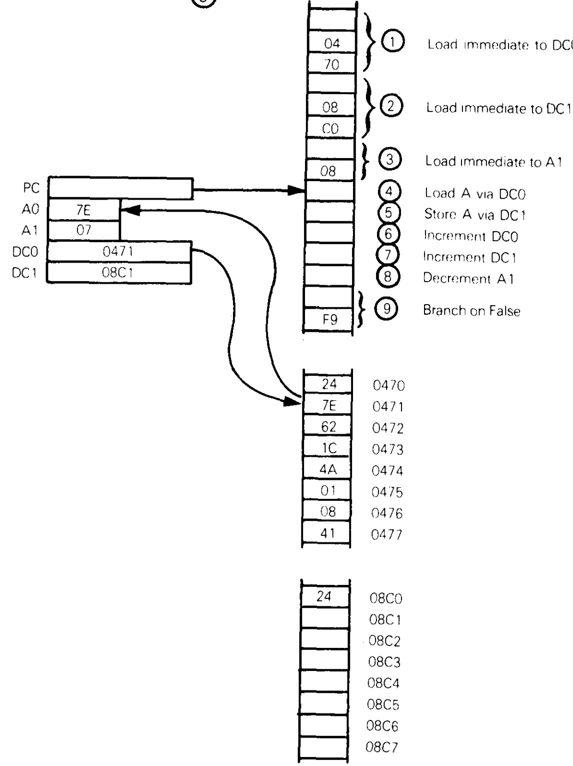
⑦


PC
A0
A1
DC0
DC1
24
08
0471
08C1
04
70
08
C0
08
F9
① Load immediate to DC0
② Load immediate to DC1
③ Load immediate to A1
④ Load A via DC0
⑤ Store A via DC1
⑥ Increment DC0
⑦ Increment DC1
⑧ Decrement A1
⑨ Branch on False
24
7E
62
1C
4A
01
08
41
0470
0471
0472
0473
0474
0475
0476
0477
24
08C0
08C1
08C2
08C3
08C4
08C5
08C6
08C7

⑧

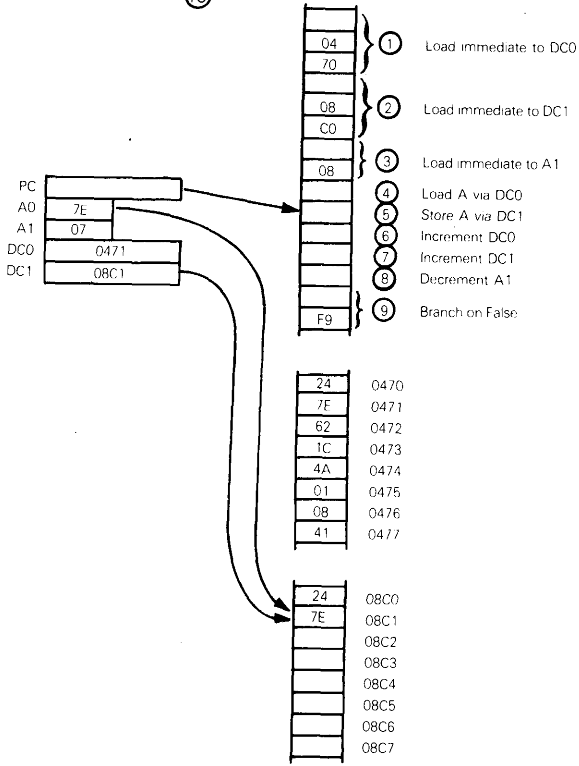

PC
A0
A1
DC0
DC1
24
07
0471
08C1
04
70
08
C0
08
F9
① Load immediate to DC0
② Load immediate to DC1
③ Load immediate to A1
④ Load A via DC0
⑤ Store A via DC1
⑥ Increment DC0
⑦ Increment DC1
⑧ Decrement A1
⑨ Branch on False
24
7E
62
1C
4A
01
08
41
0470
0471
0472
0473
0474
0475
0476
0477
08C0
08C1
08C2
08C3
08C4
08C5
08C6
08C7

(9)



PC
A0
A1
DC0
DC1
7E
07
0471
08C1
04
70
08
C0
08
F9
1 Load immediate to DC0
2 Load immediate to DC1
3 Load immediate to A1
4 Load A via DC0
5 Store A via DC1
6 Increment DC0
7 Increment DC1
8 Decrement A1
9 Branch on False
24 0470
7E 0471
62 0472
1C 0473
4A 0474
01 0475
08 0476
41 0477
24 08C0
08C1
08C2
08C3
08C4
08C5
08C6
08C7

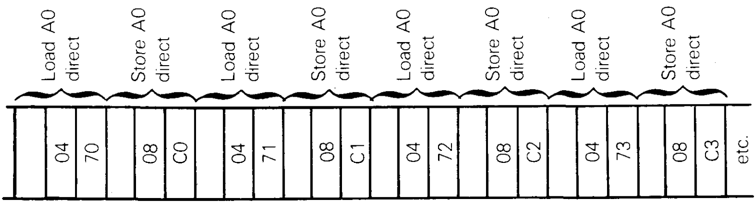
(10)



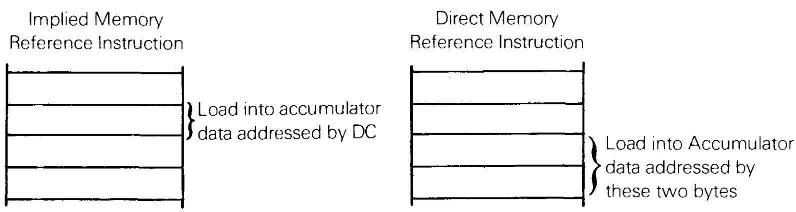
PC
A0
A1
DC0
DC1
7E
07
0471
08C1
04
70
08
C0
08
F9
1 Load immediate to DC0
2 Load immediate to DC1
3 Load immediate to A1
4 Load A via DC0
5 Store A via DC1
6 Increment DC0
7 Increment DC1
8 Decrement A1
9 Branch on False
24 0470
7E 0471
62 0472
1C 0473
4A 0474
01 0475
08 0476
41 0477
24 08C0
7E 08C1
08C2
08C3
08C4
08C5
08C6
08C7

Observe that the program, as illustrated, fits into 15 program memory bytes. Instructions 4 through 9 are re-executed eight times. This is possible, since with each re-execution, the part that changes, data memory addresses, changes in the Data Counters. This is the virtue of implied memory addressing.

How would the same program logic be implemented by a minicomputer that has direct addressing, but no implied addressing? If the program is in ROM, it will have to consist of a pair of three-byte instructions repeated eight times:



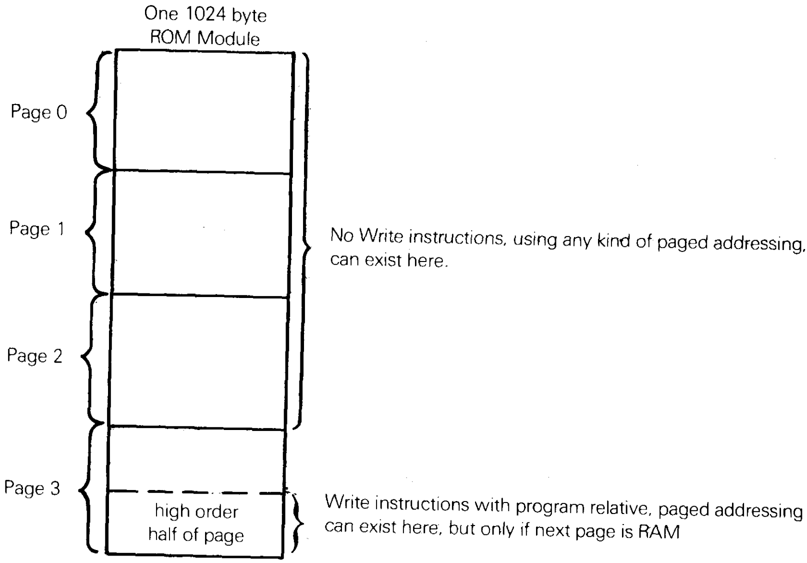
Total program length will be 48 bytes. Implied memory addressing has saved memory by allowing a set of instructions to be re-executed, and by allowing memory reference instructions to occupy a single byte:



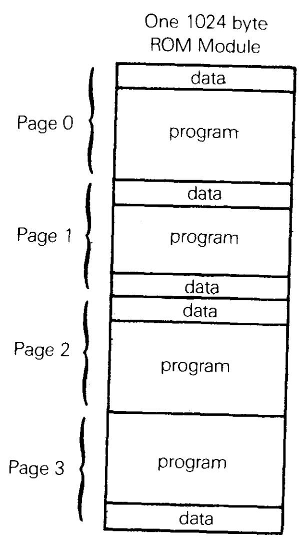
PAGED DIRECT ADDRESSING

Addressing problems associated with microcomputers become more severe if you try to use any type of paged direct addressing. Minicomputer designers use paged direct addressing in order to reduce the number of address bits that are part of each memory reference instruction. Pages may be absolute or program relative, as we have just described.

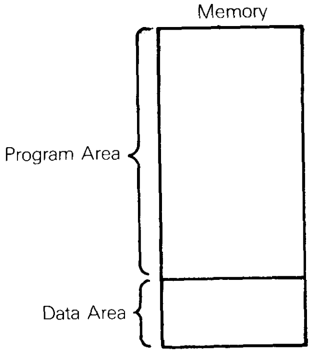
The problem with any form of paged, direct addressing is that it can only be used for Jump and Branch instructions. This form of addressing simply cannot be used for memory reference instructions that write into memory. ROM usually comes in 1024-byte (or larger) modules. Pages are either 128 or 256 bytes long. Therefore, an entire page will either be ROM or RAM. It is not possible to have the program area of a page in ROM and the data area of the same page in RAM:



It would certainly be possible to have memory reference instructions that read data out of program memory, but this would require dividing up memory into a checkerboard of program and data areas, as follows:



This type of complex memory mapping greatly increases the cost of creating microcomputer programs, plus the potential for introducing programming errors. As a result, microcomputer programs are almost always written with separate program and data areas of memory as follows:

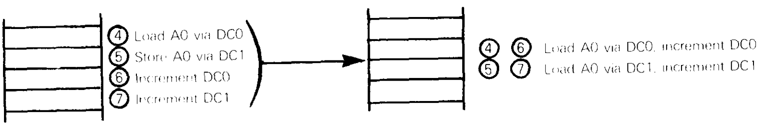


So universal is this division of memory into program and data areas that some microcomputers specify separate memories for programs and for data. This means that the Program Counter addresses a memory which the Data Counter cannot address; and conversely, the Data Counter addresses a memory which the Program Counter cannot address.

AUTO INCREMENT AND AUTO DECREMENT

In the data movement example we just described, observe that the two addresses, stored in the two Data Counters, must be incremented after each memory reference.

It is easy to imagine program logic that starts at the other end of a data buffer, so the memory address must be decremented after every memory access. In either case, **we can create a single one-byte instruction that specifies a memory reference operation, plus a memory address increment or decrement:**



THE STACK

There is a variation of implied memory addressing which has existed in many minicomputers and is implemented in one form or another in almost every microcomputer; it is known as Stack addressing. The concept of a stack was introduced at the end of Chapter 4, in connection with Chip Slice Control Unit addressing logic.

MEMORY STACKS

STACK POINTER

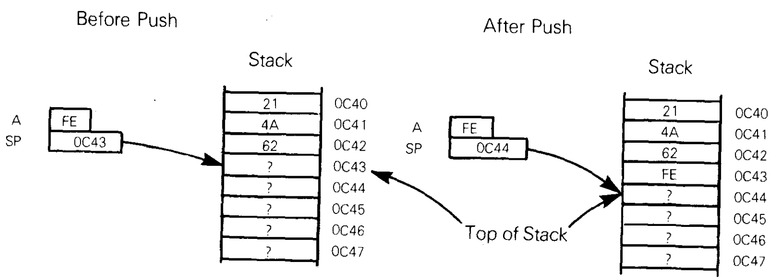
The more common Stack architecture sets aside some area of data memory for transient storage of data and addresses. The Stack is addressed by a Data Counter type of register, called the Stack Pointer.

Only two Stack operations are usually allowed: writing to the top of the Stack (referred to as a Push), and reading from the top of the Stack (referred to as a Pop, or a Pull).

The Stack gets its name from the fact that it may be visualized as a stack of data words, where only the last data word entered into the stack, or the first empty data word at the top of the stack, may be accessed. In either case the Stack is accessed via an address stored in the Stack Pointer.

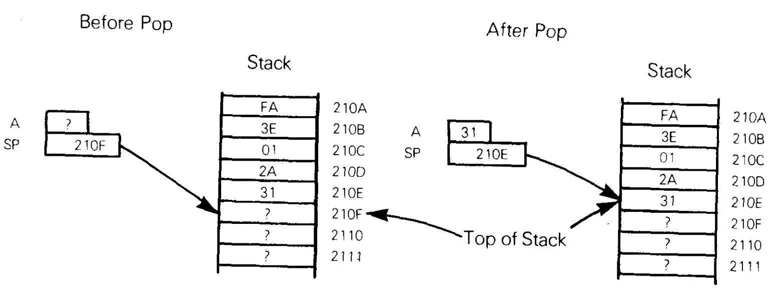
A Push operation, which writes into the Stack, will cause data from the Accumulator (or some other CPU register) to be written into the memory word currently addressed by the Stack Pointer (SP); the Stack Pointer contents is then automatically incremented to address the next free word, at the new top of the Stack, as follows:

PUSH



A Pull or Pop operation is the exact reverse of a Push; the Stack register contents is decremented to address the last word written into the top of the Stack, then the contents of the memory word addressed by the Stack is moved to the Accumulator or some other CPU register. This may be illustrated as follows:

POP



Observe that at the end of a Pop operation, the Stack Pointer is again addressing the first unused memory word at the top of the Stack; once data has been read out of the top of the Stack, the data word is assumed to be empty.

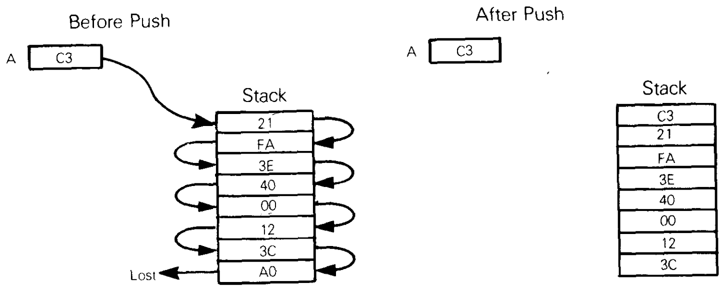
The parallel between implied memory addressing, using a Data Counter, and Stack memory addressing, using a Stack Pointer, is self-evident. In fact the only difference between the two is that the Stack Pointer contents MUST be incremented after a write, and MUST be decremented after a read.

There is, of course, nothing to stop a Stack being implemented in memory in the opposite direction; that is, the bottom of the Stack as accessed rather than the top, in the sense illustrated above. Now the Stack will be decremented after a write and incremented after a read. Otherwise nothing changes.

THE CASCADE STACK

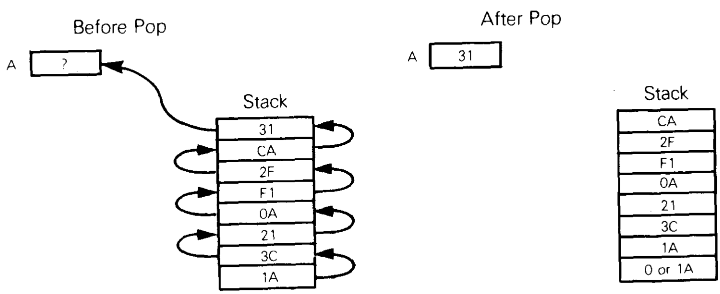
PUSH

There is an alternative Stack architecture that is less frequently seen. This architecture provides a limited number of registers (it is commonly 8 or 16) in the CPU. When a byte of data is pushed onto the Stack it cascades down as follows:



POP

When a byte of data is pulled or popped from the top of the Stack, data cascades up as follows:



This Stack architecture requires no Stack Pointer, since at all times data is being written into, or read out of the top of the Stack.

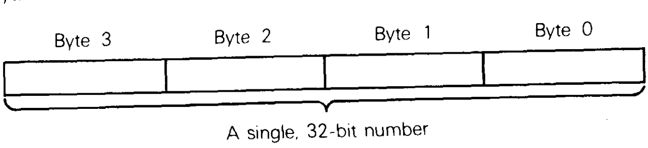
HOW A STACK IS USED

The Stack is a great convenience to minicomputer users; it is an absolute necessity in microcomputer applications.

SUBROUTINES

Consider the use of subroutines. Most programs, whether they are written for a minicomputer or a microcomputer, consist of a number of frequently used instruction sequences, each of which is recorded once, somewhere in program memory. The routine is then accessed as a subroutine.

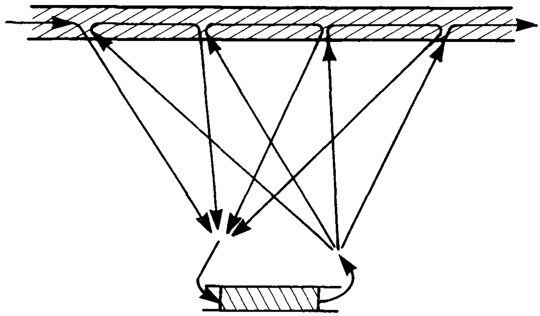
An application may require arithmetic to be performed on 32-bit numbers, occupying four contiguous bytes as follows:



The most efficient way of handling this type of arithmetic is to write four separate programs to perform 32-bit addition, subtraction, multiplication, and division. Now **every time you want to perform addition (for example), you will use an instruction which CALLS the subroutine.** A call may be illustrated as follows:

SUBROUTINE CALL

Program Memory Used by Main Program

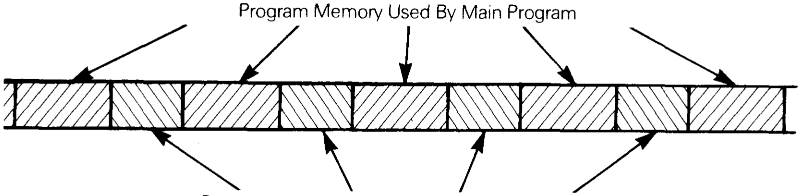


Program Memory used
by 32-bit binary
addition subroutine

——► represents the instruction execution sequence

Suppose you did not use subroutines; the instruction sequence needed to perform 32-bit, binary addition would have to be repeated every time program logic specified 32-bit, binary addition. The program would now appear as follows:

Program Memory Used By Main Program



Program Memory Used By 32-Bit, Binary Addition

Most programs, whether they are written for minicomputers or microcomputers, eventually become nothing more than a large network of calls to subroutines. Providing the importance of subroutines in all microcomputer programs is accepted at face value, you need not understand any more about subroutines at this point. However, let us consider what happens when a subroutine is called, and how program logic handles a return from a subroutine.

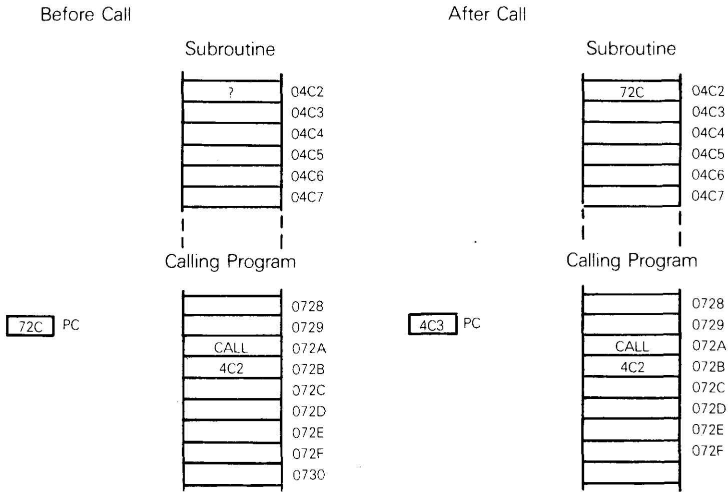
The PDP-8 minicomputer, and a number of other old minicomputers use the first memory word of a subroutine as the location where the return address is to be stored. For example, suppose our 32-bit, binary addition subroutine instructions occupy memory locations $4C2_{16}$ through $4E0_{16}$. The PDP-8, being a 12-bit minicomputer, stores only 12-bit addresses. Memory word $4C2_{16}$ is the first word of the subroutine, and must be left empty. If the subroutine is called from memory location $72A_{16}$, the following sequence of events occur:

1) The current contents of the Program Counter are stored in memory location $4C2_{16}$.

2) The address of the first subroutine instruction, $4C3_{16}$, is loaded into the Program Counter.

3) Program execution proceeds at the instruction stored in memory location $4C3_{16}$.

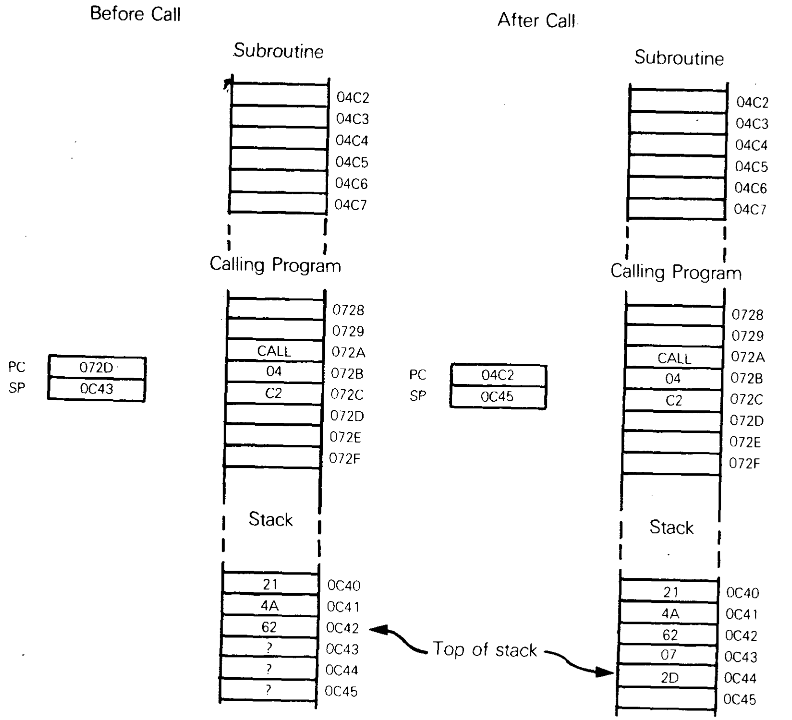
This may be illustrated as follows:



SUBROUTINE RETURN

The last instruction executed within the subroutine must be a return instruction. This instruction moves the address ($72C_{16}$) stored in the first word of the subroutine (at $04C2_{16}$) back into the Program Counter, thus causing program execution to continue at the instruction following the subroutine call.

This scheme for calling subroutines obviously cannot work in a microcomputer application, since the subroutine is going to be stored in read-only memory; this being the case, the return address cannot be stored in the first word of the subroutine. Microcomputers store subroutine return addresses in the Stack. The 32-bit binary addition subroutine call would be executed as follows:



In the above illustration, note that 8-bit memory words are assumed. Since addresses are all 16-bits long, two memory words are required to store each address.

In order to return from a subroutine, it is only necessary to pop the top two bytes of the Stack into the Program Counter. Execution will then proceed with the instruction following the call to the subroutine.

NESTED SUBROUTINES AND USE OF THE STACK

A nested subroutine is defined as a subroutine which has been called by another subroutine.

RECURSIVE SUBROUTINES

There is nothing at all unusual about one subroutine calling another. In fact, subroutines are frequently nested to a level of five or more. There are even certain mathematical routines in which the most efficient way to write the program is for the subroutine to call itself. **A subroutine that can call itself is referred to as a recursive subroutine.**

So long as the Stack is being used to preserve return addresses, subroutines can be nested in any way, or can call themselves; and providing the return path follows the call path, exactly, the correct return address will always be at the current top of the Stack.

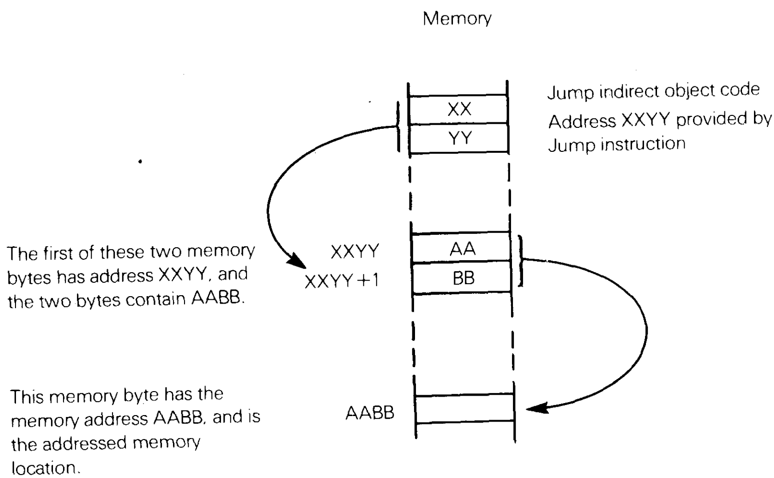
Were this book a programmer's guide, we would now prove the above statement with extensive illustrations. However, in order to understand microcomputer concepts and microcomputer programming, you can take at face value the fact that the Stack will insure that the return path is the

exact inverse of the subroutine call path. Should you still be curious, you can prove this for yourself by defining a number of subroutines located in various areas of memory. Arbitrarily select locations within subroutines where calls to other subroutines exist. Draw a Stack on a piece of paper, and for each call perform a push of the return address. For each return, pop the return address into the Program Counter. You will find that you come out of nested subroutines in exactly the same sequence as you entered them, however complex the subroutine call sequences may be.

INDIRECT ADDRESSING

INDIRECT ADDRESS

An INDIRECT ADDRESS specifies a memory word which is assumed to hold the required direct address.



INDIRECT ADDRESS COMPUTATION

For indirect addressing, the effective address is given by the equation:

EA = [XXYY]

where:

EA stands for Effective Address
[ ] signifies the contents of the memory word whose address is enclosed by the brackets.

A PAGED COMPUTER'S INDIRECT ADDRESSING

Indirect addressing is an absolute necessity on a paged computer, since it is the only way a program can access a memory location outside an instruction's own page. Thus, on a paged computer, the direct address instruction:

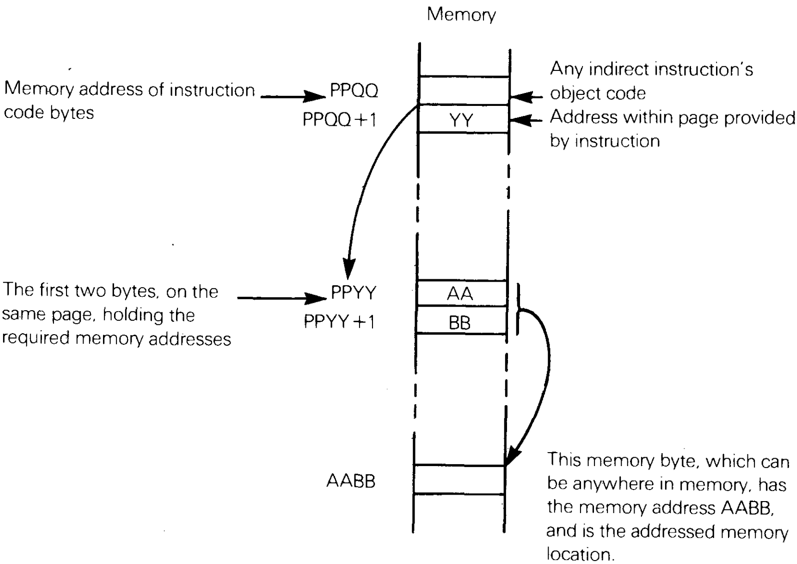
LMA XXYY

must, if XXYY is beyond the instruction's page, be replaced by the indirect address instruction:

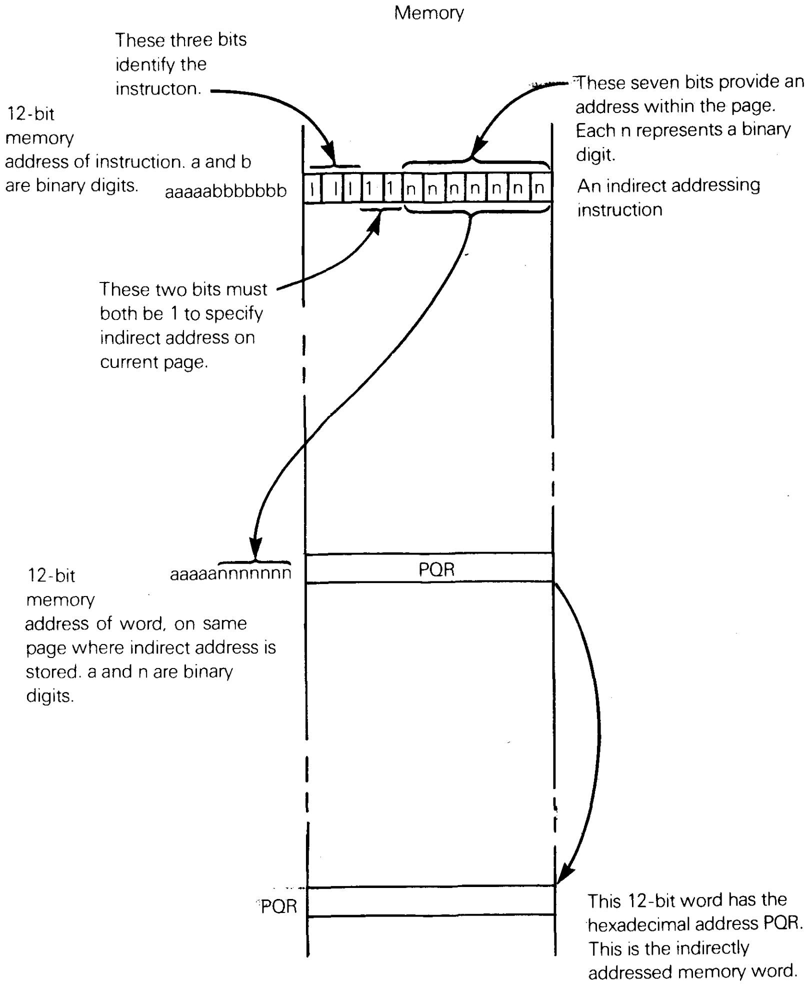
```
        LMA   *HERE
        —
        —
        —
        —
HERE    DA    XXYY
```

In the LMA (Load Accumulator) instruction, HERE provides the address of the indirect address memory word, and "*" specifies indirect addressing. Memory location HERE contains an address XXYY which *becomes the effective memory address.*

With an absolute, paged microcomputer, an instruction with indirect addressing will only occupy two bytes. The effective memory address is now computed as follows:



The PDP-8, being a 12-bit minicomputer, uses the following variation of paged, indirect addressing:



INDIRECT VIA BASE PAGE

The PDP-8, and any other paged computer that has a base page, will use a large part of the base page to store addresses; these addresses will be referenced indirectly. For example, suppose in the above illustration the object code at memory location aaaaabbbbbbb is III10nnnnnnn, instead of being III11nnnnnnn. Now the address stored in memory location 00000nnnnnnn would be chosen, not the address stored in memory location aaaaannnnnnn.

INDIRECT AUTO INCREMENT AND DECREMENT

Another variation of indirect addressing sets aside certain memory locations as auto increment or auto decrement locations. For example, the PDP-8 minicomputer sets aside memory locations 008_{16} through $00F_{16}$, on the base page, as auto increment locations. If an address is stored in any auto increment location, then the address will be incremented whenever it is referenced indirectly.

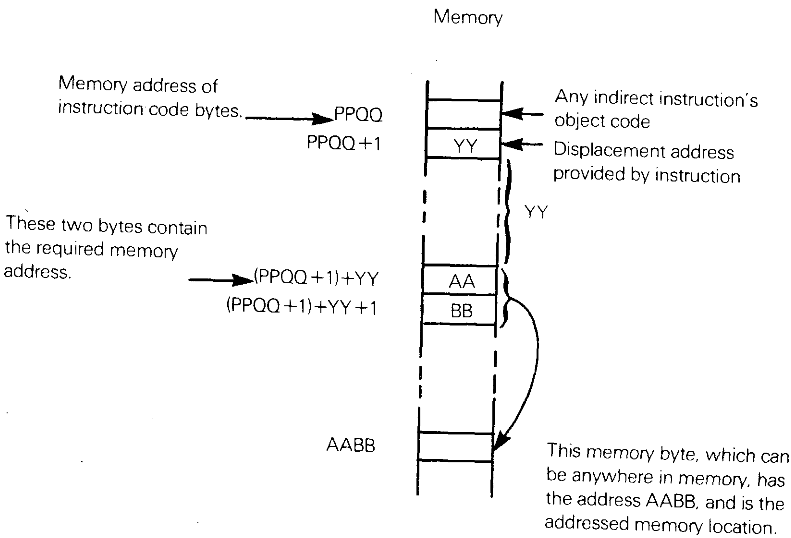
With reference to the most recent illustration, if aaaaannnnnnn were 000000001000 (i.e., 008_{16}), then after the indirect addressing instruction had executed, memory location 008_{16} would contain PQR + 1; on the next execution of the indirect addressing instruction, PQR + 1 would be the effective memory address, not PQR.

An auto decrementing indirect address would have generated PQR−1 rather than PQR+1.

Observe that the PDP-8 base page must be implemented in read-write memory if addresses stored in the base page are to change.

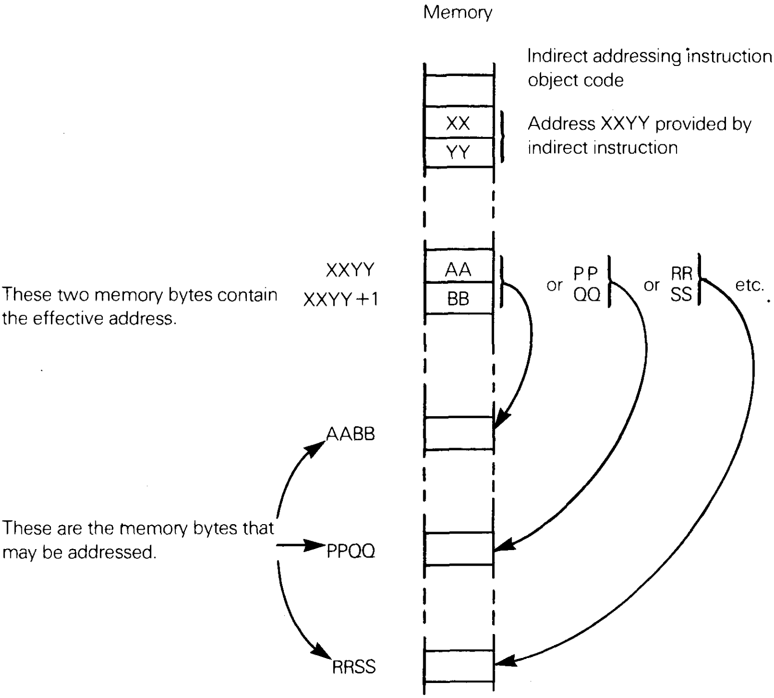
PROGRAM RELATIVE, INDIRECT ADDRESSING

A computer that uses program relative, direct addressing can also have program relative, indirect addressing. This is what happens:



INDIRECT ADDRESSING — MINICOMPUTERS VERSUS MICROCOMPUTERS

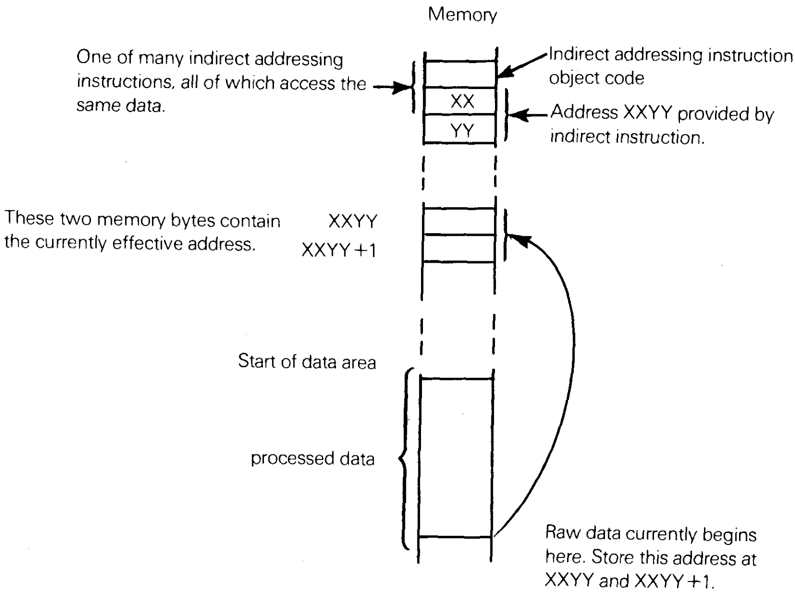
In the world of minicomputers, even if the minicomputer's direct addressing is not paged, indirect addressing is a great convenience since it allows one Load or Store instruction to access a number of different memory locations, depending on the current contents of the indirect memory address word. Consider the following example:



Why would you want to change the indirect address AABB to PPQQ or RRSS?

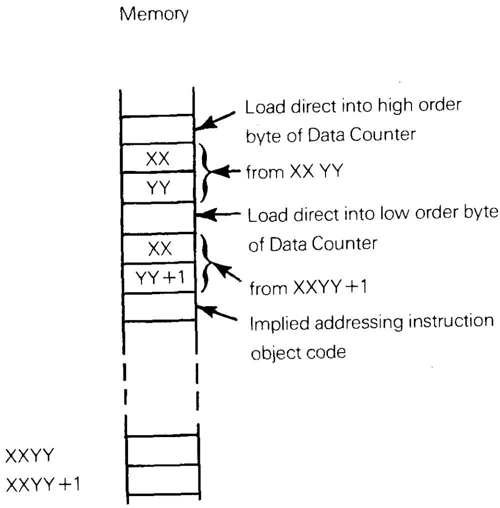
Many minicomputers are time-shared. This means that a single minicomputer may be executing many programs, attending to each one for a few milliseconds before going on to the next. Each program will use parts of memory to store programs and data that are needed for immediate execution, while the bulk of programs and data will remain on disk. A program or data table may occupy completely different areas of memory on each re-execution. This is because the area of memory that is free for use may be impacted by totally unrelated programs that were executing in preceding milliseconds by the time-sharing system. **Variable indirect addressing is one of the ways in which minicomputers are able to cope with the fact that programs and data tables may occupy different areas of memory from one execution of the program to the next.** It is only necessary to change a few addresses, such as AABB, in order to change the location of a data table or a program. **While this justification for indirect addressing makes a lot of sense in complex minicomputer applications, it makes absolutely no sense in microcomputer applications.** When an entire microcomputer system, complete with memory, costs a hundred dollars or less, it will be cheaper to give each user his own CPU and memory, rather than go to the extra programming expense required to share the use of such a low-cost item as a microcomputer.

Non-time sharing applications can also make effective use of variable indirect addressing. For example, a single data area may be used by a number of data tables of variable length. Consider a simple telecommunications application. Data is arriving over a telephone line at some random rate; as the data arrives, it is stored in read-write memory. At fixed intervals, a microcomputer program is executed to process the most recent chunk of raw data. Observe that each time the microcomputer program is executed, the data on which it must operate will reside in a different area of read-write memory. If the microcomputer program references read-write memory via indirect addressing, then by simply loading the new beginning address of the data area into the indirect address space, the microcomputer program can access the next segment of data — wherever it may be:



Of course, XXYY and XXYY+1 must be read-write memory location.

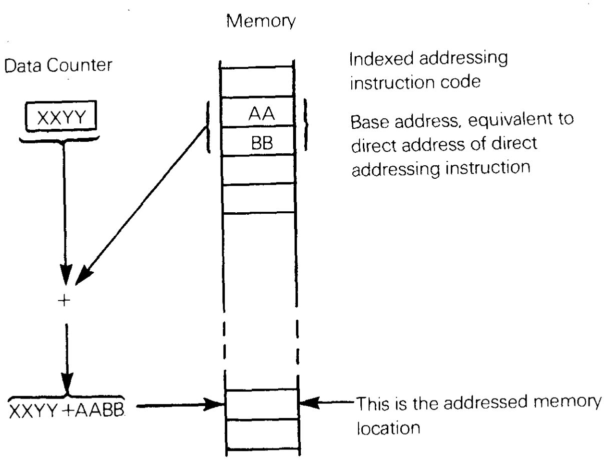
A microcomputer that has only implied and direct addressing would have to simulate indirect addressing to perform the operations described above. This could be done using three instructions for each indirect addressing instruction as follows:



INDEXED ADDRESSING

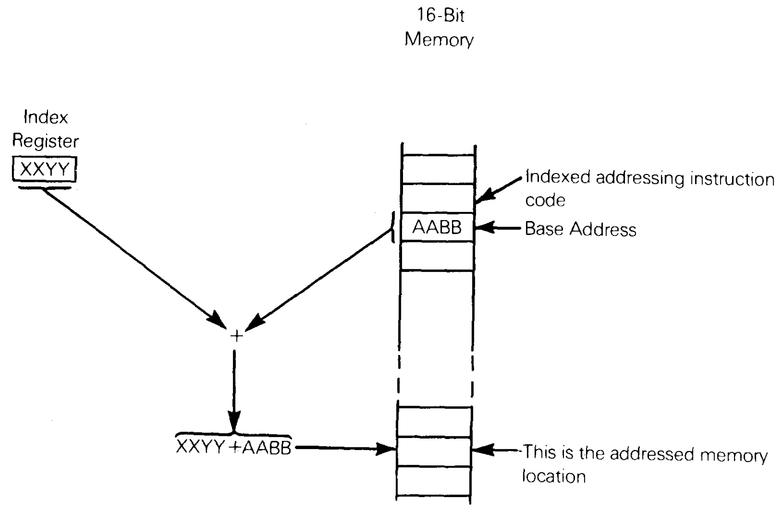
A number of microcomputer manufacturers describe implied memory addressing as indexed addressing; the two are similar but they are not the same.

An indexed address is the sum of a direct and an implied address. This may be illustratred as follows:

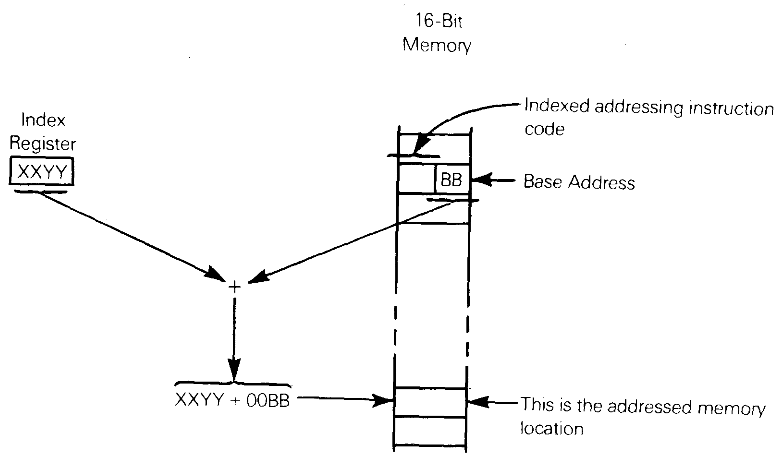


INDEX REGISTER

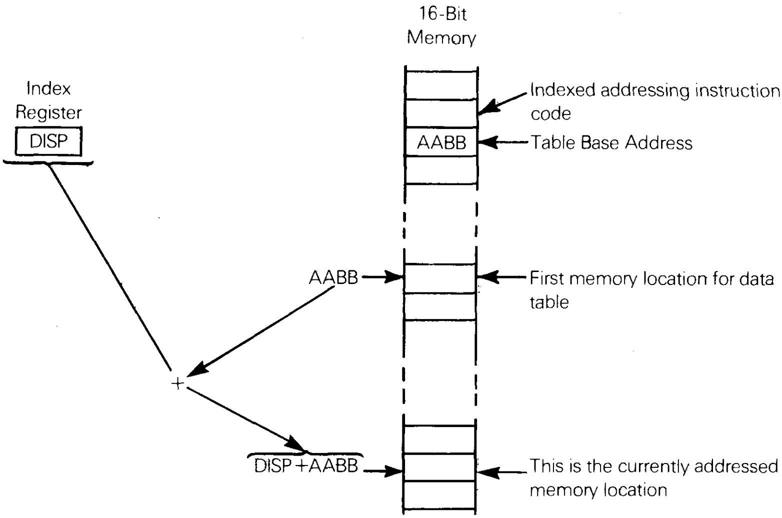
When used as illustrated above, the Data Counter is called an Index register. Some minicomputers do not have index registers, or indexed addressing; most do. Minicomputers that do have indexed addressing may have from 1 to 15 index registers. 16-bit minicomputer indexed addressing may be illustrated as follows, for long instructions:



16-bit minicomputer indexed addressing may be illustrated as follows for short instructions:

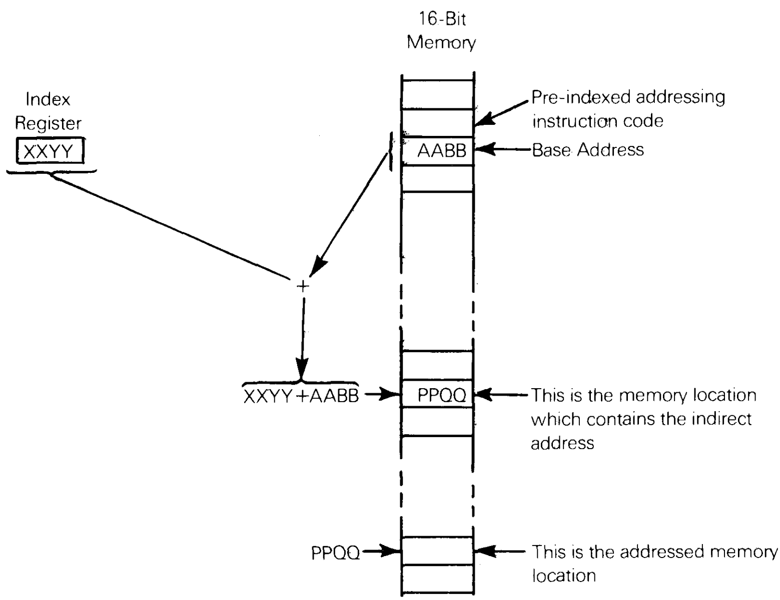


The index register derives its name from the fact that its contents is likely to change (for example, while it indexes an area of memory being treated as a data table). This may be illustrated for a 16-bit minicomputer as follows:



PRE-INDEXING

Indexed addressing may be combined with indirect addressing, and this gives rise to two possibilities; the index may be applied to the base address, or to the indirect address. When the index is applied to the base address, we talk of pre-indexed addressing. This may be illustrated as follows:



For pre-indexed addressing, the effective address (EA) is given by the equation:

EFFECTIVE ADDRESS

EA = [BASE + INDEX]

The square brackets denote "contents of". In the above illustration:

EA = [AABB + XXYY] = PPQQ

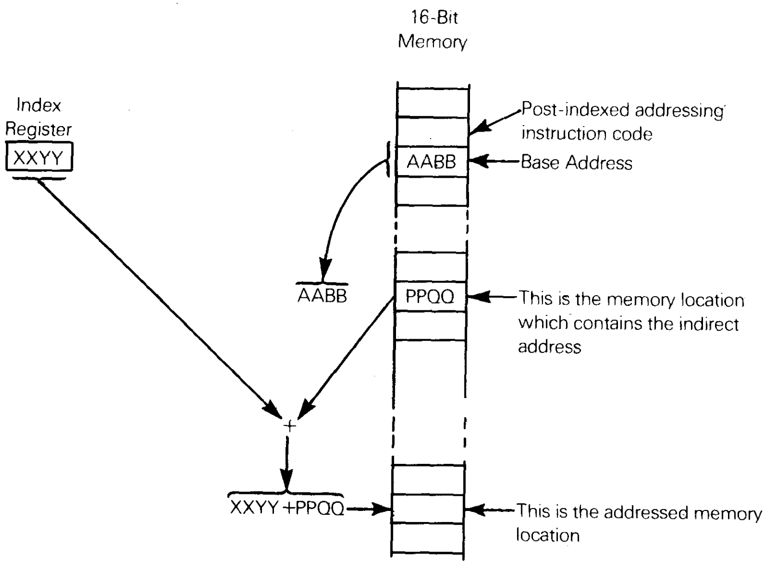
Suppose the index register contains $213A_{16}$ and the base address is $413C_{16}$, the effective address is given by:

EA = $[413C_{16} + 213A_{16}] = [6276_{16}]$

Therefore the effective address is the contents of the memory word with the address 6276_{16}.

When the index is applied to the indirect address, we talk of post-indexed addressing. This may be illustrated as follows:

POST-INDEXING



For post-indexed addressing, the effective address (EA) is given by the equation:

EFFECTIVE ADDRESS

EA = [BASE] + INDEX

Again the square brackets denote "contents of." In the above illustration:

EA = [AABB] + XXYY = PPQQ + XXYY

Again suppose the index register contains $213A_{16}$ and the base address is $413C_{16}$; the effective address is given by:

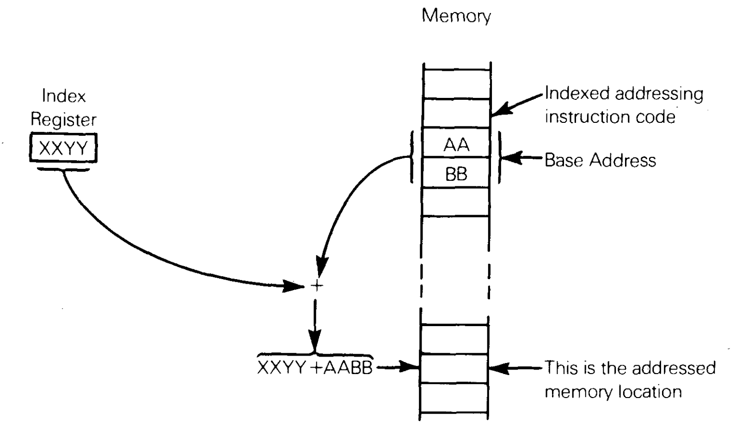
EA = $[413C_{16}] + 213A_{16}$

Therefore the effective address is the contents of the memory word with address $413C_{16}$, plus $213A_{16}$.

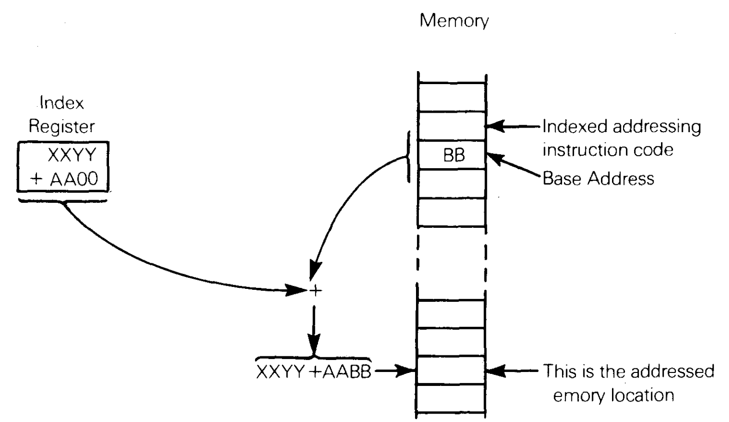
MICROCOMPUTER INDEXED ADDRESSING

An 8-bit word size is the feature of microcomputers which dictates how much indexed addressing, if any, will be implemented.

Let us begin by looking at indexed addressing in its simplest form. For an 8-bit microcomputer, this may be illustrated as follows:



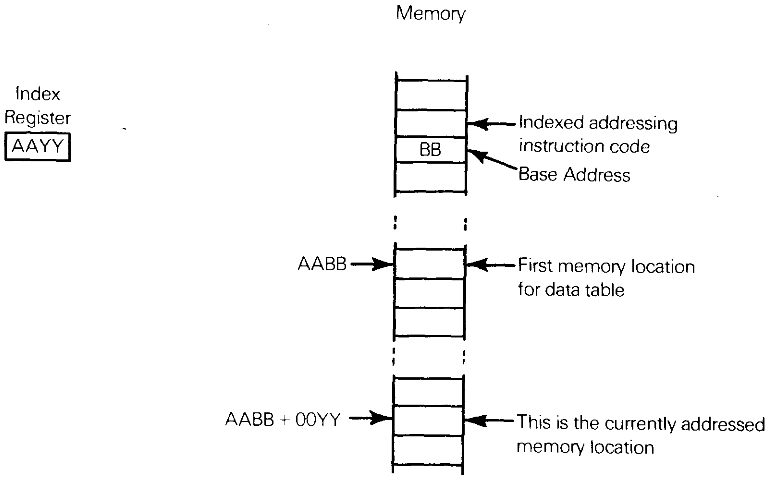
Clearly there is some redundancy in indexed addressing, as illustrated above. XXYY + AABB cannot sum to more than $FFFF_{16}$, since this is the largest value that a 16-bit address can acquire. Any valid indexed address can therefore be rewritten as follows:



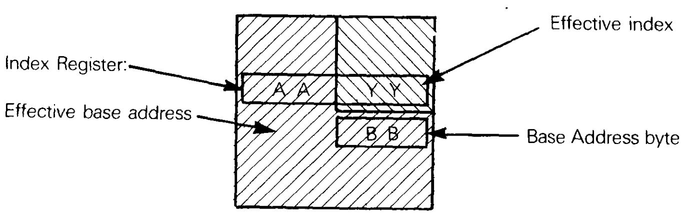
This is the effective address:

$$\begin{aligned} EA &= XXYY + AA00 + 00BB \\ &= XXYY + AABB \end{aligned}$$

We have saved a byte in our indexed addressing instruction, and given up nothing. In terms of indexing data tables, this representation of indexing may be illustrated as follows:



Since we have a 16-bit index register, but only an 8-bit memory word, what we are doing in effect, is creating the table base address out of the index register high order byte, plus the base address byte. The index register low order byte becomes the table index.



In the world of microcomputers, straightforward indexed addressing is rarely present. This is because we are dealing with an 8-bit instruction code, and if we try to specify too many addressing options, we will quickly run out of bits.

Chapter 7
AN INSTRUCTION SET

We are now ready to create a hypothetical instruction set. The instruction set we are now going to create will not copy any existing microcomputer's instruction set. Rather, it will contain features representative of all of them; what we must do is justify each feature.

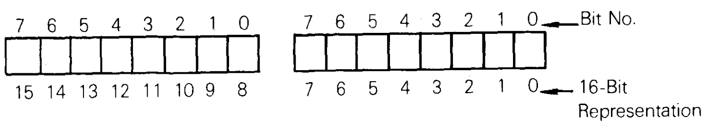
CPU ARCHITECTURE

The first prerequisite, before we can discuss individual instructions, is to select the number and type of registers, plus the number and type of addressing modes that our hypothetical miocrocomputer will have. We will start with registers.

NUMBER OF REGISTERS

We cannot simply select a large number of registers — more than enough for any situation. Remember, every register must become chip logic, using up limited real estate on the CPU chip; also, if we have many registers, we will use up many instruction code bits, simply identifying which register is to be referenced. Therefore we must carefully justify every single register we elect to have.

We are going to select two Accumulators (A0 and A1). Having two Accumulators, rather than one, simplifies 16-bit data operations, since the two Accumulators can be visualized as the upper and lower bytes of a single 16-bit unit:



16-bit data operations are seen frequently enough to justify having two Accumulators.

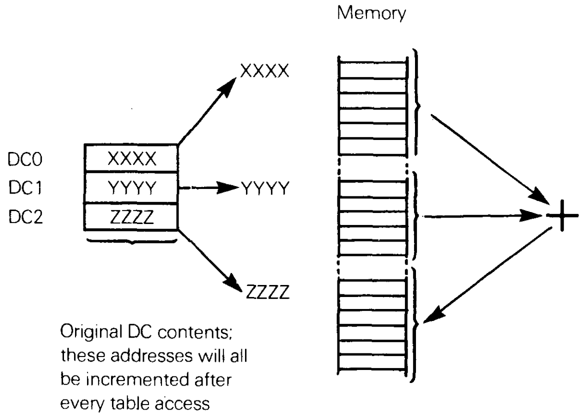
Having two Accumulators is also useful when data from two tables must be read and processed in parallel; this is easier and faster with two Accumulators, which, in effect, provide two independent channels for data transfer.

DATA COUNTERS

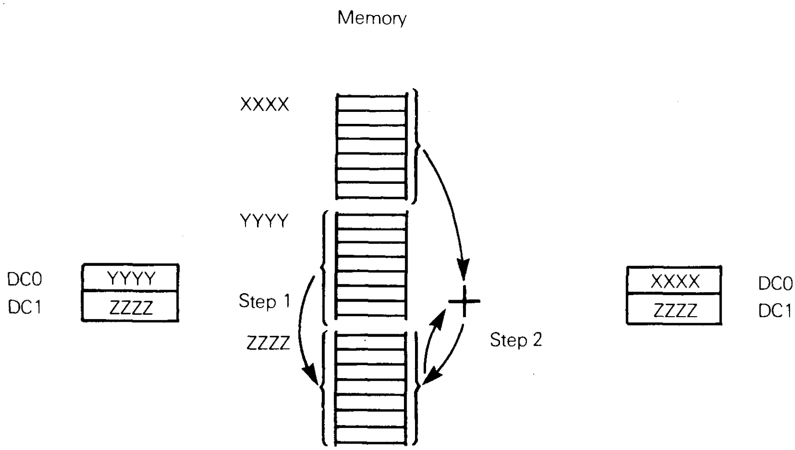
Will two Accumulators be sufficient? Some microcomputers have more than two Accumulators, or equivalent registers, but the additional registers are used as Data Counters or memory address registers of some form. **We are going to provide our microcomputer with three 16-bit Data Counters (DC0, DC1 and DC2);** therefore, we do not need more than two Accumulators, or equivalent registers.

Why have three Data Counters? The answer is that it greatly simplifies processing data out of tables. Frequently data from two source tables are combined in some way (the most obvious example is multi-byte addition) and the result is stored in a third table. Microcomputers with less than three Data Counters, or equivalent address registers must shuffle addresses between temporary storage in memory, or must otherwise circumvent the limitations of having only one (or two) Data Counters.

Consider the simple case of multibyte addition. Having three Data Counters, this operation, and similar operations, are easily handled as follows:

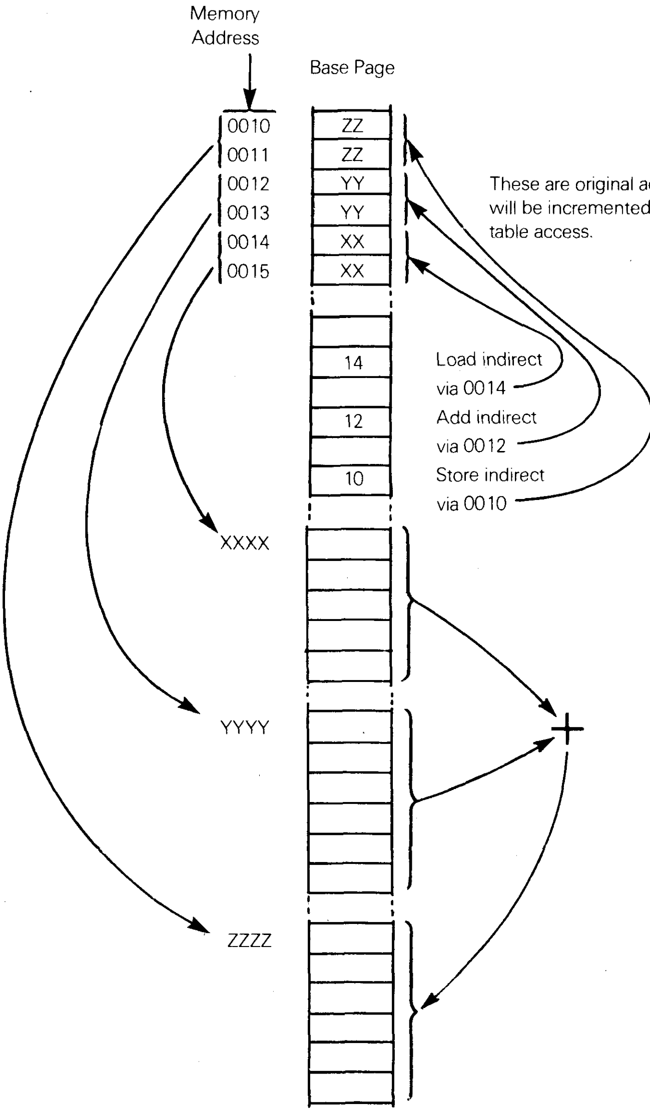


Having two Data Counters, we would have to move one of the source data buffers (beginning at either XXXX or YYYY) into the answer buffer, then add as follows:



A microcomputer with only one Data Counter would have to store the three table addresses somewhere in read-write memory, then load each address into the Data Counter before accessing each table.

A microcomputer with indirect addressing could store the three table addresses in read-write memory, then access tables indirectly via the three addresses:



CPU REGISTERS SUMMARY

We will give our microcomputer a Stack Pointer (SP) and a Program Counter (PC). Our complement of registers looks like this:

Register size	Register
8-bit	Accumulator AC0
8-bit	Accumulator AC1
16-bit	Data Counter DC0
16-bit	Data Counter DC1
16-bit	Data Counter DC2
16-bit	Stack Pointer SP
16-bit	Program Counter PC

STATUS FLAGS

In Chapter 2 we described status flags, and how they are used. We will provide our hypothetical microcomputer with the four status flags Z (Zero), C (Carry), O (Overflow) and S (Sign).

ADDRESSING MODES

A microcomputer's addressing modes will be influenced by the number and type of registers which have been selected.

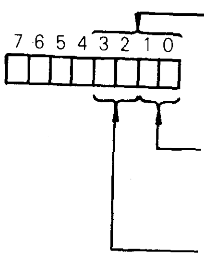
For example, a microcomputer with only *one* Data Counter is likely to provide indirect addressing as an alternative means of simultaneously accessing a number of data areas; this was illustrated in our discussion of Data Counters.

Since we have three Data Counters, we will save on instruction code bits, and CPU chip logic, by having no indirect addressing; rather, we will include auto increment and auto decrement features. **Instructions that reference memory will be divided into two categories as follows:**

1) **Load and Store instructions.** Because they are frequently used, these two instructions will have a complete, and flexible set of addressing options.

2) **Other memory reference instructions,** being less commonly used, will have a more limited memory addressing ability.

The complete memory addressing capability offered by the Load and Store instructions may be represented as follows:



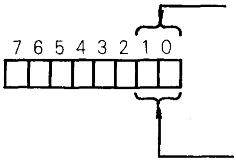
Address specification bits for a Load or Store instruction.

00 use Data Counter DC0
01 use Data Counter DC1
10 use Data Counter DC2
11 direct addressing. The next two bytes provide the direct memory address

If bits 0 and 1 are 11, these two bits are unused. If bits 0 and 1 are not 11, then interpret bits 2 and 3 as follows:

00 specifies simple, implied memory addressing
01 increment contents of specified Data Counter at conclusion of instruction
10 decrement contents of specified Data Counter at conclusion of instruction
11 increment and skip feature (described with Load and Store instructions)

Memory reference instructions, other than Load and Store, will offer this limited subset of addressing options:



Address specification bits for memory reference instructions other than Load and Store

00 Use Data Counter DC0 — Simple, implied memory addressing only
01 Use Data Counter DC1
10 Use Data Counter DC2
11 Direct addressing. The next two bytes provide the direct memory address.

A DESCRIPTION OF INSTRUCTIONS

There are two competing perspectives which we must maintain while evaluating the instructions that are to constitute our microcomputer's instruction set. First, we must decide what instruction types are vital, very useful, or simply desirable; next, we must select instructions that use the 256 possible combinations provided by an 8-bit instruction code; we cannot have more than this number of options.

In order to balance our two perspectives in the following discussion, we are going to create a complete, but hypothetical, microcomputer instruction set. This means that we must justify each instruction, or instruction type, and we must specify the object code pattern which is to be interpreted by the CPU Control Unit as identifying the specified instruction.

INPUT/OUTPUT INSTRUCTIONS

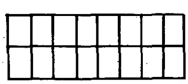
A microcomputer system would be useless if it did not provide means for receiving data from, and transmitting data to external devices; this is input/output and is specified via input/output (I/O) instructions.

An input/output instruction needs to specify three things:

1) **Is the instruction reading data from an external device (input), or is it transmitting data to an external device (output)?**
2) **As we discussed in Chapter 5, most microcomputer systems have, or at least allow, more than one port through which data can be transferred between external devices and a microcomputer system. We must identify the I/O port via which the input or output operation is to occur.**
3) **What is the source (for input) or destination (for output), within the microcomputer system, for data being transferred via I/O instructions?**

Input/output operations are so frequently used in microcomputer applications, that in order to save memory, it is a good idea to include a few single-byte I/O instructions.

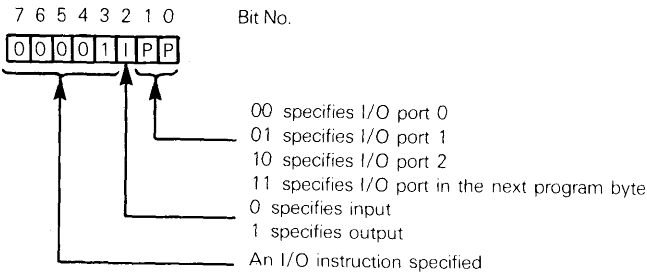
We could use just four of the 256 object code options, two for input (one for each Accumulator as the destination), the other two for output (one for each Accumulator as the source), then specify one of 256 possible I/O ports in an immediate data byte to follow:



Byte 1: Input or Output instruction.

Byte 2: Using this I/O port.

This scheme for I/O instructions is better:



We have used up eight object code options, without specifying which Accumulator is involved in the data transfer. These are the eight object code options used by I/O instructions:

00001000	Input via I/O Port 0
00001001	Input via I/O Port 1
00001010	Input via I/O Port 2
00001011	Input via I/O port addressed by next byte
00001100	Output via I/O Port 0
00001101	Output via I/O Port 1
00001110	Output via I/O Port 2
00001111	Output via I/O port addressed by next byte

It is going to take an additional bit to specify one of the two Accumulators as the data source or destination in I/O instructions. The eight object code options illustrated above would have to be repeated (perhaps with bit 4 set to 1) in order to represent two sets of I/O instructions, one set accessing Accumulator A0, the other set accessing Accumulator A1. As a result, 16 object code options would be consumed by I/O instructions; and that is unnecessarily extravagant. **Instead we will stipulate that Accumulator A0 will always be the source or destination of data for I/O instructions.**

PRIMARY ACCUMULATOR

This preferred use of Accumulator A0 will occur frequently in our instruction set, since it is an easy way of reducing the number of object code options used up by any instruction type. By making one Accumulator always more accessible, rather than spreading preference between the two Accumulators, the programmer can think and program in terms of a primary Accumulator (A0) and a secondary Accumulator (A1).

INPUT SHORT

We will use the following mnemonics for our I/O instructions:

For Input Short:

```
INS P
```

P is the instruction operand, and must be 0, 1 or 2 to specify one of the three I/O ports allowed by a single byte I/O instruction. The Assembler will flag any other value in the operand field as illegal.

INPUT LONG

Next, for Input Long:

```
IN P
```

This time P may have any value from 0 through 255. A two-byte instruction will be generated as follows:

0	0	0	0	1	0	1	1	Input to A0
								Via this I/O port.

For Output Short: **OUTPUT SHORT**

OUTS P

This is identical to the Input Short instruction, except that data will be output from Accumulator A0, through the specified I/O port (0, 1 or 2 only).

And for Output Long: **OUTPUT LONG**

OUT P

This instruction is identical to the Input Long instruction, except data will be transmitted from Accumulator A0 to an external device via any I/O port from 0 through 255.

By making I/O instructions access only Accumulator A0 as the source or destination for a data transfer, we have decided that it is more important to specify a limited number of ports within a one-byte I/O instruction, rather than allow either of the two Accumulators to be the data source or destination.

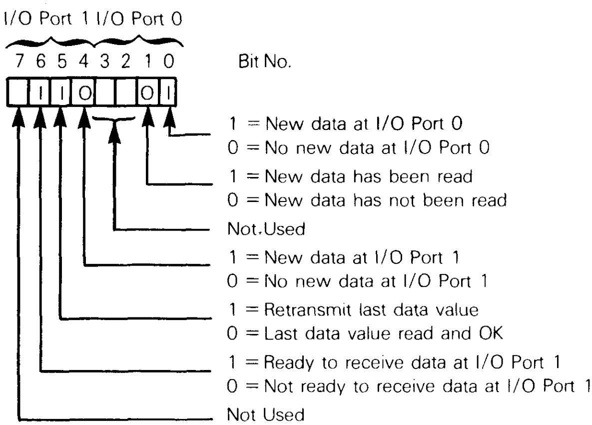
Referring to the shower temperature controlling example, let us assume that temperature readings arrive through I/O Port 0, while controls are output to the hot water valve via I/O Port 1. I/O Port 2 is used as a common status port for inputs and outputs. Information at these I/O ports will be interpreted as follows:

I/O Port 0: A millivoltage, ranging from 0 mv through 255 mv. Temperature, in °F, may be approximated as follows:

$$°F = 30 + 0.45 \text{ mv.}$$

I/O Port 1: A signed binary number, specifying the hot water valve must be opened (positive) or closed (negative). The amount of valve movement will be proportional to the absolute value 0 through 127.

I/O Port 2: Bits of this port will be interpreted as follows:



In the Port 2 illustration, I represents bits input by an external device; O represents bits output by the CPU.

MEMORY REFERENCE INSTRUCTIONS

If data arriving from a temperature sensor arrives in multibyte units, each data byte that is loaded into A0 by an input instruction must immediately be stored in read-write memory. Data output to the hot water controller must be read from memory, loaded into A0, then output via an output instruction. The data output to the hot water controller depends on the data input from the temperature sensor; in the process of computing the data to be output, any program will have to constantly reference data in memory — to load, store, add, perform logical operations, etc.

The basic architecture of any computer, mini or micro, provides a very limited data storage capacity in CPU registers, and a (relatively) enormous data storage capacity in memory external to the CPU. This makes memory reference instructions the next most vital, after I/O instructions. Recall from Chapter 5 that some microcomputers treat I/O instructions as a subset of memory reference instructions, by assigning specific memory addresses to I/O ports.

As might be expected, **the two most commonly used microcomputer memory reference instructions merely move data to or from memory; these are the Load and Store instructions.**

A Load instruction moves data from a memory location to an Accumulator.

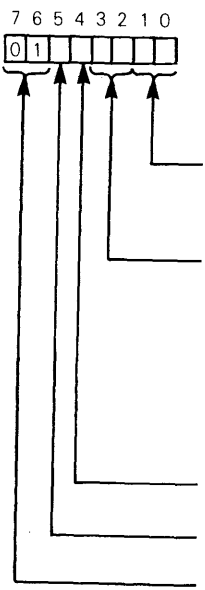
LOAD

A Store instruction moves data from an Accumulator to a memory location.

STORE

These being the two most commonly used memory reference instructions, we will spend the bits needed to include in the Load and Store instruction very flexible addressing modes.

Load and Store instruction object codes will appear as follows:



Bit No.

00 Use Data Counter DC0
01 Use Data Counter DC1
10 Use Data Counter DC2
11 Direct addressing. The next two bytes provide the direct memory address.

If Bits 0 and 1 are 11, these two bits are unused. If bits 0 and 1 are not 11, then interpret bits 2 and 3 as follows:
00 Specifies simple, implied memory addressing.
01 Increment contents of specified Data Counter at conclusion of instruction.
10 Decrement contents of specified Data Counter at conclusion of instruction.
11 Increment and skip feature

0 Reference Accumulator A0
1 Reference Accumulator A1

0 Load
1 Store

A Load or Store instruction

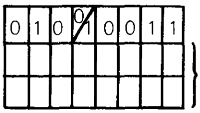
Let us now look at the complete addressing capabilities offered by the Load and Store instructions, starting with the simplest.

After describing what the addressing modes are, we will justify each one.

DIRECT ADDRESSING

A Load or Store instruction with direct addressing will have 1's in bits 0 and 1, and the direct address will be provided in the two bytes that follow. Observe that bits 2 and 3 are not used for direct addressing; they must, however, have a definite value. We will therefore specify that bits 2 and 3 must both be 0 for a direct addressing Load or Store instruction; these instructions will now have the following object code:

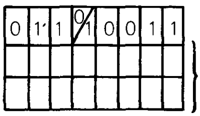
Byte 1
Byte 2
Byte 3



Load into Accumulator 0 or 1

the contents of the memory location addressed by these two bytes

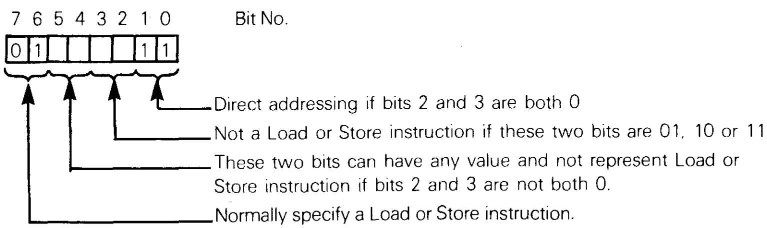
Byte 1
Byte 2
Byte 3



Store the contents of Accumulator 0 or 1

into the memory location addressed by these two bytes

The following object codes have nothing to do with Load or Store instructions:

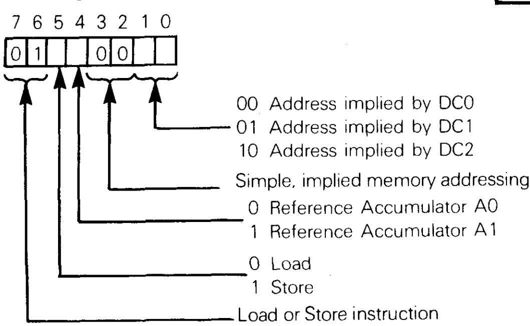


NUMBER OF LOAD AND STORE INSTRUCTIONS

There are 64 object code combinations resulting from 01XXXXXX, where X may be 0 or 1; these are the Load and Store instruction object codes. Twelve of the combinations do not represent Load or Store instructions, as illustrated above (3 combinations of bits 2 and 3, times 4 combinations of bits 4 and 5, equal 12 combinations). Therefore, there are 52 variations of the Load and Store instructions.

IMPLIED ADDRESSING

Load and Store instructions with implied memory addressing can have any of the following object codes:



The effective memory address for the Load or Store instruction is the contents of Data Counter DC0, DC1 or DC2, whichever has been specified by bits 0 and 1.

AUTO INCREMENT

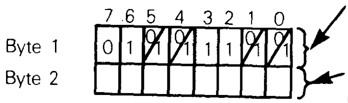
AUTO DECREMENT

Introducing the auto increment and auto decrement feature is quite easy to understand; the auto increment feature says that the implied memory address, that is, the contents of the specified Data Counter, will be incremented by 1 at the conclusion of the memory reference instruction. Conversely, the auto decrement feature specifies that the Data Counter contents will be decremented by 1 at the conclusion of the instruction.

The object code for an auto increment instruction will differ from the implied addressing equivalent object code only in bits 3 and 2, which will be 01; the equivalent auto decrement instruction will have 10 in bits 3 and 2.

INCREMENT AND SKIP

Increment and skip, specified by having 1's in bits 2 and 3, is not a common microcomputer feature. We are going to create Load and Store instructions with the following format:



Load or Store, then increment contents of the Data Counter specified by bits 0 and 1; then check contents of Data Counter.

If the last 6 bits of the Data Counter contents are 000000, increment the Program Counter to bypass (and ignore) this byte. If the last 6 bits of the Data Counter have any other value, treat this byte as a signed binary number, to be added to the Program Counter contents, forcing a Branch.

The most effective way of illustrating the necessity for the various addressing modes is with short program sequences. Let us therefore first describe the instruction mnemonics which will be used for the Load and Store instructions.

LOAD DIRECT

STORE DIRECT

Load and Store Direct will use these mnemonics:

LRA	ADDR	Load direct into A0
LRB	ADDR	Load direct into A1
SRA	ADDR	Store direct from A0
SRB	ADDR	Store direct from A1

ADDR is any symbol representing a memory location from which data will be read or to which data will be written. We use the letter A to represent A0 and B to represent A1; we could use the digits 0 and 1, but it is too easy to confuse 0 and O, and 1 with l; therefore, use of 0 and 1 within instruction mnemonics is unpopular.

LOAD IMPLIED

STORE IMPLIED

Load and Store Implied will use these mnemonics:

LMA	DCX	Load into A0 from the memory location addressed by DCX
LMB	DCX	Load into A1 from the memory location addressed by DCX
SMA	DCX	Store the contents of A0 into the memory location addressed by DCX
SMB	DCX	Store the contents of A1 into the memory location addressed by DCX

DCX specifies one of the three Data Counters and therefore must be DC0, DC1 or DC2.

The Load or Store with Auto Increment or Auto Decrement instructions will be identical to Load/Store Implied, as described above, except that the specified Data Counter contents will be incremented or decremented. We will use the instruction mnemonics LNA and LNB for Load with Auto Increment, LDA and LDB for Load with Auto Decrement, SNA and SNB for Store with Auto-Increment and SDA and SDB for Store with Auto Decrement.

LOAD/STORE WITH AUTO INCREMENT OR DECREMENT

Load and Store instructions with Auto Increment and Skip will be specified by the following instruction mnemonics:

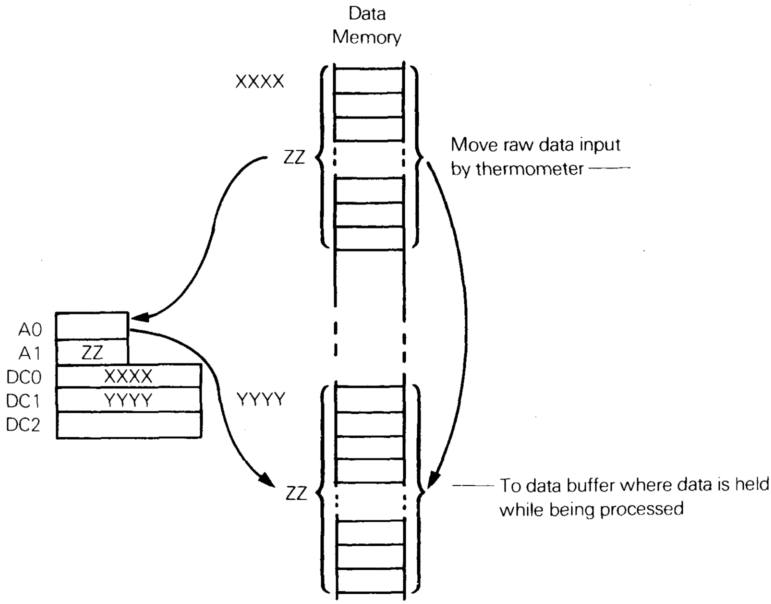
LOAD/STORE WITH AUTO INCREMENT AND SKIP

LSA	DCX,LABEL	Load into A0
LSB	DCX,LABEL	Load into A1
SSA	DCX,LABEL	Store contents of A0
SSB	DCX,LABEL	Store contents of A1

DCX identifies the Data Counter holding the implied memory address; it must therefore be DC0, DC1 or DC2.

LABEL is a symbol identifying the instruction which will be executed next if, after DCX is incremented, the last six binary digits are not all zeros.

In order to demonstrate the power of the Load and Store instructions, let us look at a simple problem which moves data from one buffer to another. Assuming that the starting addresses of the source and destination buffers are in Data Counters DC0 and DC1, and assuming that the buffer length is stored in Accumulator A1, the problem may be illustrated as follows:



XXXX is the beginning address of the input data buffer.

YYYY is the beginning address of the output data buffer.

ZZ is the length of each data buffer.

The following instruction sequence will perform the required data move:

```
LOOP  LMA  DC0   Load next input data byte
      SMA  DC1   Store in next destination buffer byte
      Increment DC0 contents
      Increment DC1 contents
      Decrement A1 contents
      If A1 contains 0, branch to LOOP
```

Instructions which we have not yet described are written out in words, rather than using unfamiliar instruction mnemonics.

AUTO INCREMENT OR DECREMENT JUSTIFICATION

Now we will introduce the auto increment feature, and this is what happens to our instruction sequence:

```
LOOP  LNA  DC0   Load next input data byte.
                 Increment address.
      SNA  DC1   Store in next destination buffer byte. Increment address.
      Decrement A1 contents
      If A1 contains 0, branch to LOOP
```

Two instructions have been removed from the six instruction sequence and two bytes of object program code have been saved.

AUTO INCREMENT AND SKIP JUSTIFICATION

Now assume that the destination buffer ends at memory location $08C0_{16}$; the last six binary digits of this address are all zeros:

$$08C0_{16} = 0000100011\underbrace{000000}_{\text{tested for auto-skip}}$$

We can now compress our data move program to these two instructions:

```
LOOP  LNA  DC0        Load next input data byte; increment address.
      SSA  DC1,LOOP   Store, increment and skip on end.
```

These two instructions occupy three bytes, as follows:

0	1	0	0	0	1	0	0	Load A0 via DC0; increment DC0
0	1	1	0	1	1	0	1	Store A0 via DC1; increment DC1 and skip
1	1	1	1	1	1	1	0	Twos complement of 2

We no longer need to hold the buffer length in Accumulator A1. Nor do we need to decrement the buffer length, or increment memory addresses. After the Store, Increment and Skip instruction increments the destination buffer address, it tests the incremented value; if the incremented value does not end in six binary zero digits, execution will return to the Load instruction; this two-instruction loop will be continuously re-executed until the Store, Increment and Skip instruction does increment the destination address to $08C0_{16}$; at this point the branch will be bypassed and the instruction which immediately follows the above data movement loop will be executed.

A minicomputer programmer would recoil at an addressing scheme such as the auto increment and skip. The idea that data tables must be placed at memory addresses ending in six binary zeros poses more problems than it offers advantages.

While the minicomputer programmer may see the neatness of instruction loops that require no special end-of-loop logic, problems associated with data relocation would be horrible: if a program were to be re-used in another application, or if it were part of a time-sharing system, the programmer would constantly have to worry about ensuring that data tables ended at correct memory boundaries — or else the program would simply not work.

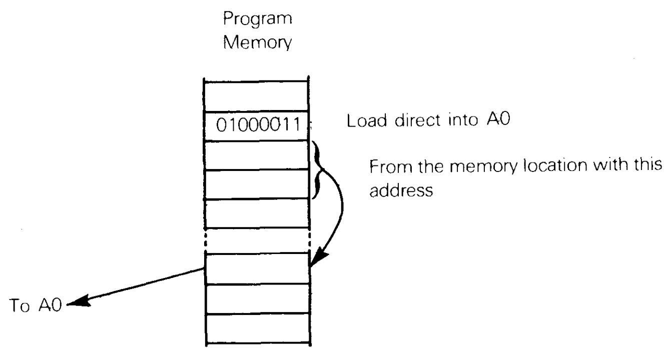
We return once again to the old minicomputer programmer's axiom: "Remember that whatever you do today may impact tomorrow." But remember that in the world of microcomputers there is no tomorrow. Whatever you do today becomes a ROM chip and will never again change. Mapping data tables onto memory address boundaries is only a minor inconvenience, since memory mapping will be a significant part of every microcomputer programming assignment anyway. When the ROM chips that result from your program may be reproduced thousands of times, you will want to be absolutely sure that your program resides in the fewest, smallest chips possible.

The auto increment and skip feature offers very significant advantages in a microcomputer application because it saves on object program bytes, while the penalty paid — having to map data tables onto address boundaries — is part of a job that must be done in any event.

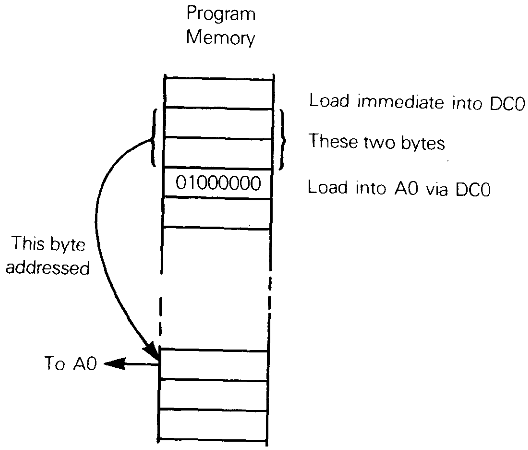
DIRECT ADDRESSING JUSTIFICATION

We have not yet justified the need for direct addressing instructions. Are they necessary?

There is nothing a direct addressing instruction does which could not be done with an implied addressing instruction; in certain cases, however, direct addressing instructions use less memory. Consider the buffer length which we were going to load into Accumulator A1, and then decrement. In the end we eliminated this logic sequence from our data movement example, but there are going to be many instances in which this type of logic cannot be eliminated. How does the buffer length, or any similar number, get loaded into the Accumulator? A three-byte direct addressing instruction will do the job as follows:



Using implied memory addressing the operation will require four bytes and will temporarily use a Data Counter as follows:

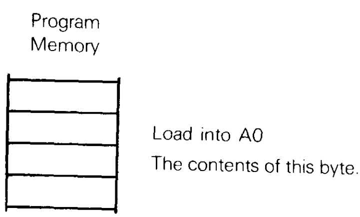


The three-byte immediate instruction which is needed to load data into a Data Counter is an unnecessary expense, when it is followed by a single memory reference, such as the one-time load of an index into another register; as we have seen earlier in this chapter, the three bytes needed to load an address into a Data Counter result in a great subsequent memory savings, but only if the address in the Data Counter is going to be re-used many times.

If a microcomputer is to have either direct addressing or implied addressing, implied addressing is the more desirable; for example, the Intel 8008, which was the predecessor of the Intel 8080, has implied addressing but no direct addressing.

Most programs load single values (such as counters and indexes) into registers frequently enough to make direct addressing justifiable.

Note that if the counter or index to be loaded into a register has a value that will never change, you would use neither direct nor implied addressing to load the value into a register. You would use immediate addressing:



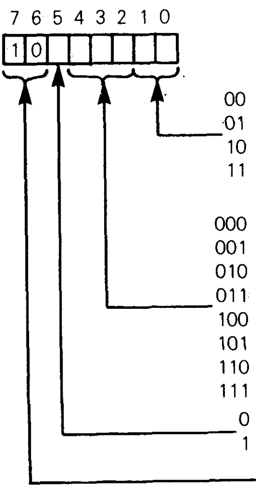
SECONDARY MEMORY REFERENCE (MEMORY REFERENCE OPERATE) INSTRUCTIONS

Let us now look at memory reference instructions other than Load and Store. In each case an operation will be performed using the contents of one of the Accumulators, plus the contents of an addressed memory location. The result will always be stored in the identified Accumulator, erasing whatever value had previously been in the Accumulator (the Compare instruction described below is an exception). The Zero, Sign, Carry and Overflow status flags will be set or reset to reflect the result of the operation. For example, the "Add memory to A1" instruction will add the contents of the addressed memory location to the contents of Accumulator A1. The previous contents of memory are not changed.

With the exception of the Store instruction, microcomputer instructions will avoid modifying memory since that implies the presence of read-write memory.

ADD
ADD DECIMAL
SUBTRACT DECIMAL
AND
OR
XOR
COMPARE

We will include these secondary memory reference instructions:



00 Use Data Counter DC0 — Simple, implied memory addressing only
01 Use Data Counter DC1
10 Use Data Counter DC2
11 Direct addressing. The next two bytes provide the direct memory address.

000 Add binary
001 Add decimal
010 Subtract decimal
011 AND
100 OR
101 Exclusive OR
110 Compare
111 Not Used

0 Use Accumulator A0
1 Use Accumulator A1

Secondary memory reference instructions

These are the instruction mnemonics we will use:

ABA	or	ABB	Add Binary to A0 or Al
ADA	or	ADB	Add Decimal to A0 or A1
DSA	or	DSB	Decimal Subtract from A0 or A1
ANA	or	ANB	AND with A0 or A1
ORA	or	ORB	OR with A0 or A1
XRA	or	XRB	Exclusive OR with A0 or A1
CMA	or	CMB	Compare A0 or A1 with memory

The code "10" in bits 7 and 6 specifies that the remaining six bits represent secondary memory reference instructions. However, only seven of the eight combinations possible for bits 2, 3 and 4 are used for secondary memory reference instructions. Therefore, only 56 of the 64 bit combinations are, in fact, secondary memory reference instructions.

In each case, the instruction will be written out like this:

```
MNEM    DCX
```

where MNEM is one of the mnemonics listed above (e.g., ADA), and DCX is one of the Data Counters (DC0, DC1 or DC2).

The direct memory referencing version of the instruction will look like this:

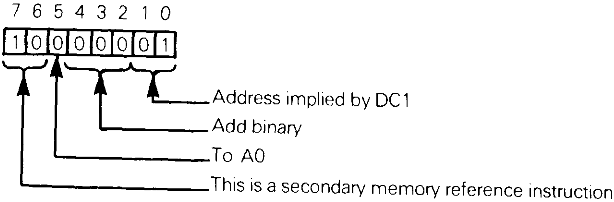
```
MNEM    ADDR
```

where ADDR is the direct address.

Here are two examples. The instruction:

```
ABA    DC1
```

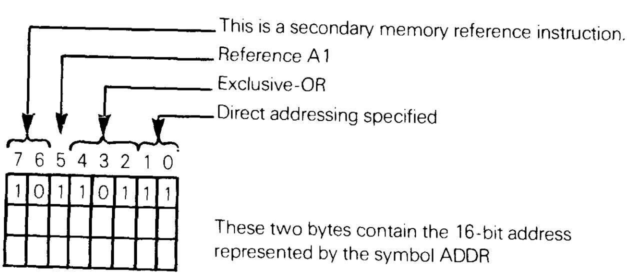
adds to A0, using binary addition, the contents of the memory location whose address is implied by DC1. This is the object code generated:



The following is a direct memory referencing instruction:

```
XRB    ADDR
```

The contents of A1 is Exclusive-ORed with the contents of the memory location addressed by symbol ADDR. These three object program bytes will be created:



Only two of the seven secondary memory reference instructions described above need any comment.

Add and subtract decimal perform decimal addition or subtraction, using three or two binary steps, as described for binary-coded decimal arithmetic in Chapter 2.

We perform decimal subtraction using a separate instruction, since the logic sequence is sufficiently different from a decimal add to make the extra instruction worthwhile.

BINARY SUBTRACT

We do not provide a separate binary subtract instruction, since this is simply a twos complement followed by an add, as described in Chapter 2.

DECIMAL ADJUST

Note carefully that the decimal addition and subtraction instructions are not the same thing as a decimal adjust instruction. Decimal adjust simply takes the contents of a register and re-arranges the bits to create the decimal equivalent of the binary number; this is explained in Chapter 2. A majority of microcomputers offer a decimal adjust instruction; a minority offer decimal arithmetic instructions.

COMPARE

The Compare instruction subtracts the contents of the addressed memory location from the specified Accumulator; the result of the subtraction is discarded — it is not stored in the specified Accumulator. However, the Z status bit is set or reset to reflect the result of the subtraction. This is a very useful instruction since it allows the program execution sequence to be determined by the relative magnitude of data items.

The Branch on Condition instructions, described later in this chapter, take advantage of, and are used in conjunction with the Compare instruction.

SECONDARY MEMORY REFERENCE INSTRUCTIONS JUSTIFICATION

Are the secondary memory reference instructions necessary? This question must be answered two ways:

First, are the operations performed by the secondary memory reference instructions necessary?

Second, must these operations be performed using secondary memory reference instructions?

The operations described — addition, Boolean logic and compare, are such basic steps in any logic sequence that a microcomputer that did not offer these logic capabilities, one way or another, would be worthless.

There is, however, no reason why these operations have to be part of memory reference instructions. For example, it would be possible to load the two operands into Accumulators A0 and A1, then to perform the same operations, register-to-register. Microcomputers which have many Accumulators, such as the Intel 8080, favor register-to-register instructions over register-to-memory instructions; microcomputers with fewer Accumulators, such as the Motorola M6800, use register-to-memory instructions as we have described.

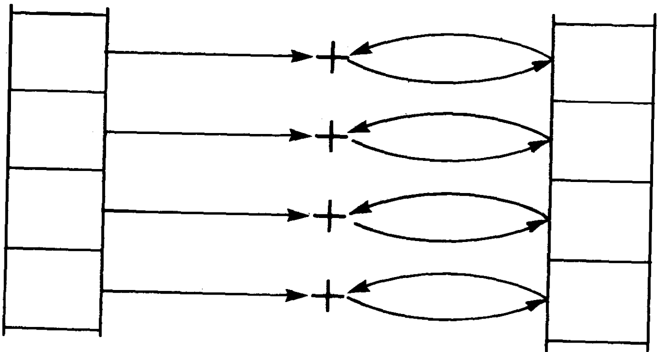
Let us look at a simple example.

MULTIBYTE ADDITION

Suppose the two buffers beginning at XXXX and YYYY (in the preceding illustration) each hold a single, multibyte number. The number in the buffer beginning at XXXX could be added to the number in the buffer beginning at YYYY as follows:

```
      Initially clear the carry status
LOOP  LMA  DC0       Load the next input byte
      ABA  DC1       Add binary from answer buffer
      SSA  DC1,LOOP  Store the result, increment and skip
```

The above three-instruction program assumes that the buffer beginning at YYYY (this address is stored in DC1) contains one of the numbers to be added, but at the end of the addition this buffer will contain the answer. This logic works, since the answer is going to over-store the byte which was just added; therefore, information is destroyed only when it will not be needed in the future. This may be illustrated as follows:



Another three instruction loop can perform a binary addition where the result is stored in a third buffer, which we will assume is addressed by DC2. The three instructions will look like this:

```
LOOP  LNA  DC0        Load next augend byte
      ABA  DC1        Add corresponding addend byte
      SSA  DC2,LOOP   Store the result, increment and skip
```

An example of the usefulness of the Boolean secondary memory reference instructions is to test switches for setting changes.

BOOLEAN LOGIC JUSTIFIED

SWITCH CHANGE TESTS

Suppose the status of eight switches are input to I/O Port 4; the previous settings for these eight switches are stored in memory at a location addressed by the symbol SWITCH. The following instruction sequence identifies which switches have changed settings and how the settings have changed:

```
IN   4        Input new switch settings
XRA  SWITCH   Identify changed switches
Save contents of A0 in A1
ANA  SWITCH   Identify switches that turned on
```

This is what the above three instructions do:

The first instruction inputs the new switch settings to A0; suppose these settings are:

01100101

Where 0 represents an "off" switch and 1 represents an "on" switch.

Suppose the previous switch settings, stored in the memory location identified by the symbol SWITCH, are:

10101101

Switches 7 and 3 were "on", and are now "off". Switch 6 was "off", and is now "on". Switches 5, 4, 2, 1, and 0 have not changed.

EXCLUSIVE-OR

The XRA instruction leaves the Exclusive-OR of the old and new switch settings in A0:

Old Settings:	10101101	
New Settings:	01100101	
XRA:	11001000	gives changed switches

The changed switches are identified by 1 bits, and are stored in A1.

AND

The AND instruction leaves the AND of the old switch settings, and the changed switch settings in A0:

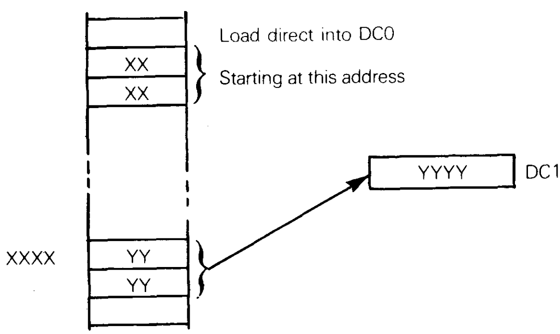
Changed Settings:	11001000	
Old Settings:	10101101	
ANA:	10001000	gives on-to-off switches.

LOAD IMMEDIATE INSTRUCTIONS, JUMP AND JUMP-TO-SUBROUTINE

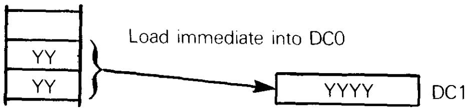
The concept of immediate addressing has been discussed frequently as a means of loading data or addresses into registers. How vital are immediate addressing instructions to a microcomputer instruction set?

IMMEDIATE INSTRUCTIONS JUSTIFICATION

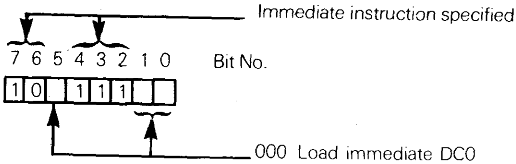
We cannot use implied memory addressing to load an address into a Data Counter, since implied memory addressing requires a Data Counter to already hold an address. Direct addressing could do the job. A base address stored in two memory bytes could be directly addressed, and loaded into a Data Counter, as follows:



But the above illustration clearly has some redundant bytes; the address being loaded into the Data Counter could just as easily be stored in the two direct memory address bytes as follows:



Immediate instructions are not absolutely vital to a microcomputer instruction set, but they are certainly a great convenience; we will therefore include eight immediate addressing instructions: to load data into the three Data Counters, the Stack Pointer, the Program Counter (with two variations) or the two Accumulators. These will be either two- or three-byte instructions; since the Accumulators are only one byte long, immediate instructions that load data into an Accumulator will be followed by just one byte of immediate data. The Data Counters, the Program Counter and the Stack Pointer are all two bytes long; therefore, immediate instructions that load data into any of these registers will be followed by two bytes of immediate data. **The following object code patterns will specify immediate instructions:**



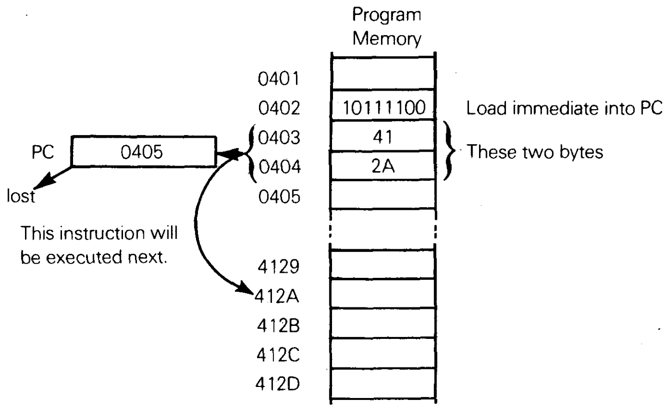
000 Load immediate DC0
001 Load immediate DC1
010 Load immediate DC2
011 Load immediate SP
100 Load immediate PC (Jump)
101 Load immediate PC (Jump to subroutine)
110 Load immediate A0
111 Load immediate A1

Since there are eight immediate instructions, and there were eight unused object code combinations from within the secondary memory reference instruction group, we use these eight unused combinations for immediate instructions, as illustrated above.

JUMP INSTRUCTION

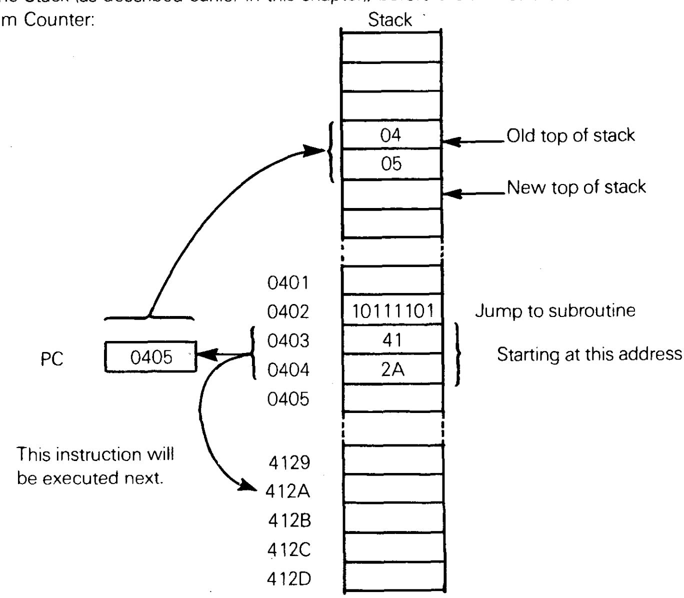
Special attention must be given to the two instructions which load immediate data into the Program Counter; unlike the other immediate instructions, these two modify the program execution sequence; the next instruction executed is going to be fetched from the memory location whose address was loaded immediately into the Program Counter.

In its simplest form, this is an Unconditional Jump (or Branch) instruction:



A Jump-to-Subroutine instruction differs only in that the current Program Counter contents must be saved before the new immediate data is loaded into the Program Counter; since our microcomputer has a Stack, the Program Counter contents will be pushed onto the Stack (as described earlier in this chapter), before the immediate data is loaded into the Program Counter:

JUMP TO SUBROUTINE INSTRUCTION



Most microcomputers' instruction set descriptions do not include Jump and Jump-to-Subroutine instructions in the immediate instruction category; instruction logic is, however, almost identical.

Instruction mnemonics for immediate instructions will be different for Jump and for the straightforward Load Immediate instructions. **For Load Immediate we will use the following mnemonics:**

LOAD IMMEDIATE

```
        LIM     R,DATA
```

R must be A0, A1, DC0, DC1, DC2 or SP. DATA must be a number, or a symbol representing a number; it must be equivalent to an 8-bit value if R is A0 or A1, it must be equivalent to a 16-bit value otherwise.

The Jump instruction will appear as follows:

JUMP

```
        JMP     ADDR
```

ADDR must be the label of the instruction which is to be executed next.

The Jump-to-Subroutine instruction will appear as follows:

JUMP TO SUBROUTINE

```
        JSR     SNAME
```

SNAME must be the label of the first instruction executed within the subroutine.

We will now create a subroutine.

Let us return to the data move program that illustrated the Increment-and-Skip instruction; written out fully, this program would include additional instructions to load addresses into Data Counters, as follows:

SUBROUTINES

```
BUFA    EQU     XXXX
BUFB    EQU     YYYY
        —
        —
        —
        LIM     DC0,BUFA    Load Source initial address
        LIM     DC1,BUFB    Load Destination initial address
LOOP    LNA     DC0         Move data from Source
        SSA     DC1,LOOP    to Destination
```

The EQU mnemonics represent Assembler Equate directive.

Recall that an Assembler directive is not an instruction, and it generates no object code; instead it provides the Assembler with information without which the Assembler could not generate an object program.

EQUATE ASSEMBLER DIRECTIVE

EQU identifies an Equate directive; this directive tells the Assembler that wherever the symbol in the label field occurs, the number in the operand field must be substituted. For example, it tells the Assembler:

"Use the hexadecimal value XXXX wherever you see the symbol BUFA."

We could just as easily rewrite our program as follows:

```
        LIM     DC0,XXXX
        LIM     DC1,YYYY
LOOP    LNA     DC0
        SSA     DC1,LOOP
```

ASSEMBLER DIRECTIVES — THEIR VALUE

The advantage of having the Equate is that the symbol BUFA (or BUFB) may appear many times within a program. If the value associated with the symbol changes, all you have to do is change one Equate in the source program. When you re-assemble the source program, every reference to the changed symbol will be corrected in the new object program which the Assembler creates.

Without the Equate directive, you would have to find every source program instruction that references the changed symbol, then you would have to correct each source program instruction, with no guarantee that you found them all.

JUMP TO SUBROUTINE

To turn the data moving program into a subroutine, all we do is give a label to the instruction which is to be executed first, and to add an instruction which executes a return from the subroutine:

```
MOVE    LIM    DC0, BUFA    Load source initial address
        LIM    DC1,BUFB     Load destination initial address
LOOP    LNA    DC0          Move data from source to
        SSA    DC1,LOOP     destination
        Return from subroutine
```

The Return-from-Subroutine instruction is described with the Stack instructions; we will ignore the Return logic for now.

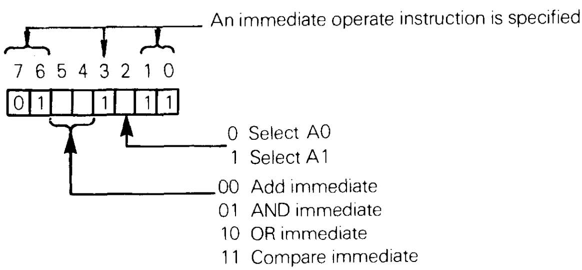
Other programs can call the subroutine with this one instruction:

```
JSR    MOVE
```

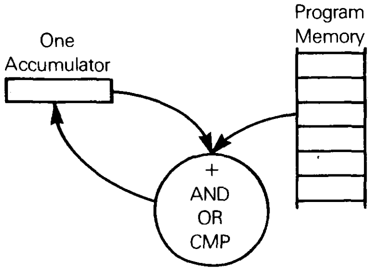
Subroutine MOVE can be called by any other program, any number of times.

IMMEDIATE OPERATE INSTRUCTIONS

A limited number of Immediate Operate instructions will be very useful; these instructions will perform operations on the contents of an Accumulator plus the immediate operand, storing the result back in the specified Accumulator. Consider these instructions:



Each instruction describes an operation that will be performed using the contents of an Accumulator, and the byte following the instruction code:



Immediate Operate instruction

The status flags C, O, Z and S will be set or reset to reflect the results of the operation.

Observe that we have used eight of the twelve unused object code combinations from the 64 Load/Store patterns 01XXXXXX. These four combinations still remain unused within this pattern:

01000111
01010111
01100111
01110111

These four combinations may be represented by: 01XX0111.

ADD IMMEDIATE
AND IMMEDIATE
OR IMMEDIATE
COMPARE IMMEDIATE

We will use the following mnemonics for the Immediate Operate instructions:

AIA	DATA	Add immediate to A0
AIB	DATA	Add immediate to A1
NIA	DATA	AND immediate to A0
NIB	DATA	AND immediate to A1
OIA	DATA	OR immediate with A0
OIB	DATA	OR immediate with A1
CIA	DATA	Exclusive-OR immediate with A0
CIB	DATA	Exclusive-OR immediate with A1

In each case, DATA is a number (or a symbol) that becomes an 8-bit value. In each case, two bytes of object code will be generated. For example, the OIB instruction will create this object code:

Byte 1	0	1	1	0	1	1	1	1	OR immediate with A1
Byte 2									the data in this byte

IMMEDIATE OPERATE INSTRUCTIONS JUSTIFIED

We will now demonstrate the value of the Immediate Operate instructions. Look again at how, in the I/O instruction description, Port 2 was defined as a combination control and status port.

These two instructions determine if there is new data at I/O Port 0:

INS	2	Input status
NIA	H'01'	Mask out all but the 0 bit

H'01' means 01 hexadecimal.

The NIA instruction resets to 0 all bits in Accumulator other than bit 0:

		A0 Contents
INS	2	XXXXXXXX
NIA	H'01'	0000000X

X represents either 0 or 1.

If the result is zero, bit 0 must have been zero, and no new data is at I/O Port 0; if the result is not zero, bit 0 must have been 1, so there is new data at I/O Port 0.

Recall that the Z status will record whether the NIA instruction generates a zero or a non-zero result.

After reading data from I/O Port 0, the program can reset bit 0 of the I/O Port 2 to 0, and can set bit 1 to 1, using these four instructions:

```
INS    2       Input status
NIA    H'FE'   Clear bit 0
OIA    H'02'   Set bit 1 to 1
OUTS   2       Return the result
```

This is what happens:

		A0 Contents	I/O Port 2 Contents
INS	2	XXXXXXXX	XXXXXXXX
NIA	H'FE'	XXXXXXX0	XXXXXXXX
OIA	H'02'	XXXXXX10	XXXXXXXX
OUTS	2	XXXXXX10	XXXXXX10

Again X represents any binary digit (0 or 1).

If you are unclear on how the AND and OR work, refer again to Chapter 2. All we are doing is ANDing I/O Port 2 contents with 11111110, then ORing the result with 00000010.

BRANCH ON CONDITION INSTRUCTIONS

Up to this point, status flags Zero (Z), Carry (C), Overflow (O), and Sign (S) have been useless curiosities, because the microcomputer provides no way to take advantage of the status flags.

What is the logical way of using status flags?

The answer is to provide instructions which allow program execution sequence to depend on the condition of a status flag.

We have already seen two examples of how status flags can determine the subsequent course of program's execution. In the Immediate Operate instruction description, bit 0 of I/O Port 2 is tested for a zero or non-zero value. If this bit has a zero value, program execution must branch to some instruction sequence which does not attempt to read new data from I/O Port 0. If this bit is 1, the program execution sequence must branch to a routine which will input data from I/O Port 0.

The discussion of Load and Store instruction categories started out with a routine that loads buffer length into Accumulator A1, then decrements the contents of A1 as a means of testing if the last buffer byte has been moved; so long as A1 has not decremented to zero, program

execution returns to the beginning of the data move loop; as soon as the contents of A1 decrements to zero, program execution must continue and not branch back:

```
        LIM    A0,LENGTH    Load buffer length
        LIM    DC0,BUFA     Load source buffer starting address
        LIM    DC1,BUFA     Load destination buffer starting address
LOOP    LNA    DC0          Load next input data byte, increment DC0
        SNA    DC1          Store next input data byte, increment DC1
        AIB    H'FF'        Add H'FF' to A1; this decrements A1
        If A1 does not contain 0, return to LOOP
        If A1 does contain 0, continue with next instruction
```

While the AIB instruction (which has already been described) in effect decrements the contents of A1, a Register Operate instruction (which has not yet been described) does the job in one byte, instead of two.

BRANCH ON CONDITION INSTRUCTION JUSTIFICATION

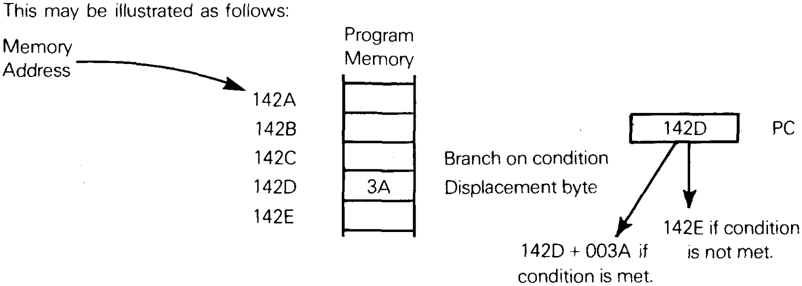
Branch on Condition instructions are vital to a microcomputer because they are the means of testing status flags; status flags in turn are vital to a microcomputer because they are the means for determining what happends when an instruction executed with more than one possible result.

There are two philosophies concerning Branch on Condition instructions; one uses Branches, the other uses Skips.

BRANCH PHILOSOPHY

A Branch on Condition instruction, as the name implies, has a one- or two-byte displacement following the instruction code (just like immediate data). If a specified condition is met, then the displacement is added to the contents of the Program Counter as a signed binary number, and thus a program branch is executed. If the specified condition is not met, the Program Counter is incremented beyond the displacement bytes and the next sequential instruction is executed.

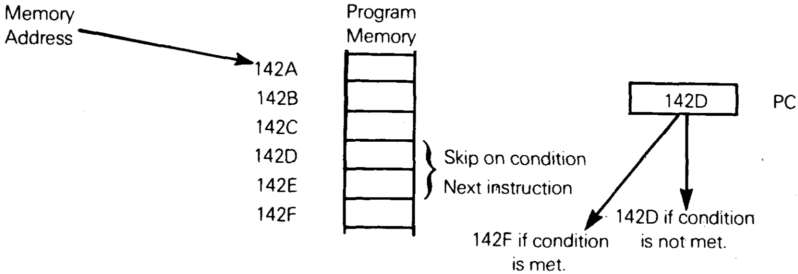
This may be illustrated as follows:



SKIP PHILOSOPHY

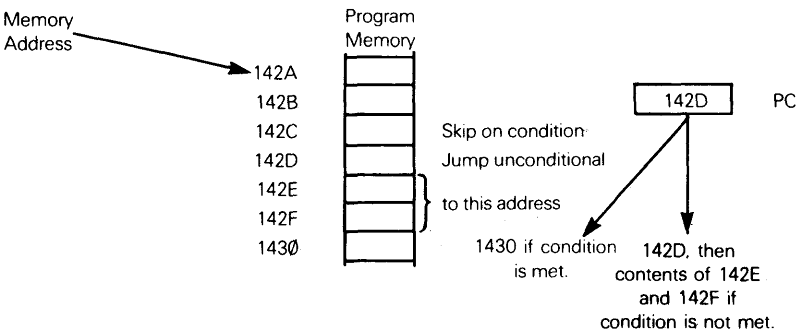
The Skip on Condition instruction has no following address displacement. The logic of this instruction states that if the specified status conditions are met, then the next sequential in-

struction will be skipped; if the specified status conditions are not met, then the next sequential instruction will be executed. This may be illustrated as follows:



In the illustration above, observe that "Next instruction" happens to be a two-byte instruction; were it a one-byte instruction, the Program Counter would be incremented to $142E_{16}$, if the condition was met. Were "Next instruction" a three-byte instruction, the Program Counter would be incremented to 1430_{16} if the condition was met.

By including an Unconditional Jump instruction directly after a skip on condition instruction, you create the inverse of a Branch on Condition instruction:



Observe that the auto increment and skip logic available with Load and Store instructions is a form of Skip on Condition instruction.

Should our microcomputer include Branch on Condition or Skip on Condition instructions? We will choose Branch on Condition instructions because they are a little more economical with this type of two-way execution sequence:

Data Movement Program Loop

```
LOOP  LNA  DC0
      SNA  DC1
      AIB  H'FF'
      Branch to Loop if A1=0

        Branch Logic
```

```
LOOP  LNA  DC0
      SNA  DC1
      AIB  H'FF'
      Skip next instruction if A1 is not
      equal to 0
      JMP  LOOP

      Skip Logic
```

What are the conditions on which we will branch? We will choose the following eight branch conditions:

BRANCH ON WHAT CONDITIONS?

Branch on Zero (Z) equals 0
Branch on Zero (Z) equals 1
Branch on Carry (C) equals 0
Branch on Carry (C) equals 1
Branch on Overflow (O) equals 0
Branch on Overflow (O) equals 1
Branch on Sign (S) equals 0
Branch on Sign (S) equals 1

Branch on Condition instructions will be followed by single-byte displacements, which means that a forward or reverse displacement of +127 or −128 bytes is possible. This is reasonable since 90% or more of all branches will be served by this range, so to provide two-byte displacements would be wasteful of memory. Of course, you can always generate a longer range branch by combining an Unconditional Jump with a Branch on Condition instruction as follows:

```
        Branch on Z = 0
        Displacement to THERE. Out of range!
```

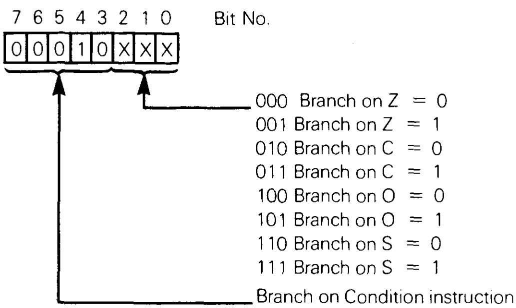
Substitute:

```
        Branch to HERE on Z = 1
        JMP     THERE
HERE    Next instruction
```

The JMP instruction is followed by a two-byte address, so it can continue execution anywhere in memory.

The Branch and Jump instruction sequence illustrated above has the same logic as a Skip on Condition instruction.

We will use the following eight object codes for our eight Branch on Condition instructions:



Branch on Condition instructions will have the format:

```
    OP    LABEL
```

LABEL is the label of the instruction to be executed if the condition specified by OP is met.

The Assembler will convert LABEL into a displacement by subtracting the current Program Counter contents from the 16-bit address value assigned to LABEL; if the result is out of range, the Assembler will print an error message.

OP will be a mnemonic as follows:

BZ	Branch on Zero (Z = 1)
BNZ	Branch on No Zero (Z = 0)
BC	Branch on Carry (C = 1)
BNC	Branch on No Carry (C = 0)
BO	Branch on Overflow (O = 1)
BNO	Branch on No Overflow (O = 0)
BP	Branch on Positive (S = 0)
BN	Branch on Negative (S = 1)

BRANCH ON LESS, EQUAL OR GREATER

The Compare instruction causes novice programmers a great deal of confusion. "Branch on zero" and "Branch on carry" are not nearly as meaningful as "Branch if greater" or "Branch if less."

Recall that the Compare instruction subtracts an operand from the contents of the specified Accumulator, and sets status flags based on the result of the subtraction. The following conditions can therefore be identified:

Branch on Accumulator less than or equal (BLE).
Branch on Accumulator less than operand (BL).
Branch on Accumulator and operand equal (BE).
Branch on Accumulator and operand not equal (BNE).
Branch on Accumulator greater than operand (BG).
Branch on Accumulator greater than or equal (BGE).

Depending on whether the Accumulator contents is being interpreted as signed or unsigned binary data, the qualitative conditional branches can be determined by using the following Boolean logic:

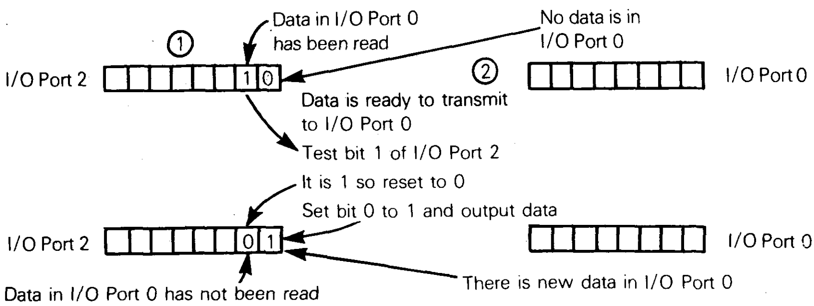
Branch Condition	Boolean Condition	
	Signed Data	Unsigned Data
BLE	Z OR (S XOR O) = 1	C = 0 OR Z = 1
BL	S XOR O = 1	C = 0
BE	Z = 1	Z = 1
BNE	Z = 0	Z = 0
BG	Z OR (S XOR O) = 0	C = 1 OR Z = 0
BGE	S XOR O = 0	C = 1

In the table above, for unsigned data, some microprocessors invert the Carry status following a subtract or compare operation. In that case, you must exchange C = 1 and C = 0.

In order to illustrate the use of Branch on Condition instructions, we will take another look at how the shower temperature controller might read data being input by the thermometer.

When the thermometer is ready to output a byte of data, it tests bit 1 of I/O Port 2. If this bit is 1, thermometer logic assumes that any previous data it sent has been read and processed; therefore thermometer logic transmits a byte of data to I/O Port 0 and signals this event by setting bit 0 of

I/O Port 2 to 1. Thermometer logic will also reset bit 1 of I/O Port 2 to 0, since the data at I/O Port 0 has not yet been read:



In order to read data input by the thermometer, the microcomputer program must keep testing bit 0 of I/O Port 2 until this bit is read as 1. Then the microcomputer must read the data in I/O Port 0; but at the same time the microcomputer must reset bit 0 of I/O Port 2 to 0 since, as soon as data is read out of I/O Port 0, it becomes old data. The program must now set bit 1 of I/O Port 2 to 1; this tells thermometer logic that the data in I/O Port 0 has been read.

The following instruction sequence performs the operations described above; in addition, this instruction sequence assumes that the data byte read out of I/O Port 0 will be stored in a memory location addressed by Data Counter DC0. Auto increment addressing is used with DC0 so that this Data Counter automatically addresses the next free byte of the input data buffer, ready for the next access of I/O Port 0.

```
LOOP    INS     2       Input status
        NIA     H'01'   Clear all bar 0 bit
        BZ      LOOP    Return to LOOP if 0 bit is 0
        INS     2       New data is ready. Input status again.
        NIA     H'FE'   Reset bit 0 to 0
        OIA     H'02'   Set bit 1 to 1
        OUTS    2       Restore the new status to I/O Port 2
        INS     0       Input the data byte
        SNA     DC0     Store in memory using implied, auto increment addressing
```

JUMP TO SUBROUTINE ON CONDITION

A number of microcomputers have Jump-to-Subroutine instructions akin to the Branch on Condition instructions we have just described. Our microcomputer has one Jump-to-Subroutine instruction which was described as an Immediate instruction.

Conditional Jump-to-Subroutine instructions will usually be followed by a two-byte address, since subroutines may well reside in memory a long way away from the Jump-to-Subroutine instructions. The logic of Conditional Jump-to-Subroutine instructions is otherwise similar to the Branch on Condition: if the specified condition is met, the Jump-to-Subroutine occurs; if not, the next instruction is executed.

CONDITIONAL RETURN FROM SUBROUTINE

Many minicomputers also have a set of Conditional Return-from-Subroutine instructions. These instructions restore to the Program Counter the address which the Jump-to-Subroutine instruction saved. We have no special Return-from-Subroutine instruction; we will use a Pop instruction instead (described along with the Stack instructions).

REGISTER-REGISTER MOVE INSTRUCTIONS

There are two types of instructions that reference two CPU registers: instructions that move data from one register to another, and instructions that perform secondary memory reference type operations, but entirely within the CPU.

REGISTER TO REGISTER MOVE INSTRUCTIONS JUSTIFIED

Register-to-register data movement instructions can be quite limited in our microcomputer, given its register organization.

We must be able to move data between A0 and A1. Exchanging the contents of the Accumulators is also frequently useful.

Moving data from the Accumulators to the Data Counters allows program logic to create variable addresses in the Accumulators, then move these addresses to a Data Counter, for variable implied addressing. Moving data in the reverse direction allows a Data Counter to be used as temporary storage for data in the Accumulators; of course, this assumes that the Data Counter in question is not being used for implied addressing.

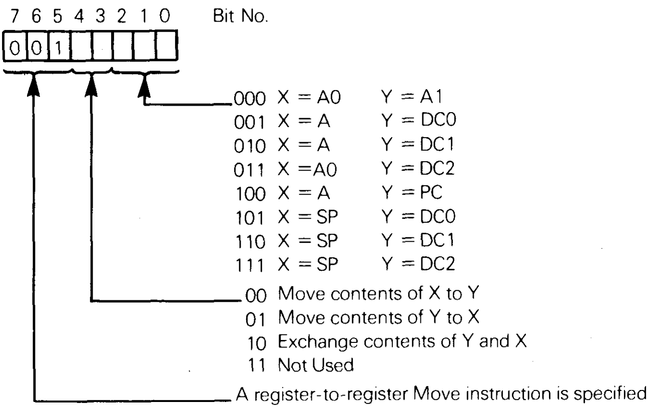
MULTIPLE STACKS

There is rarely any need to move data from one Data Counter to another. However, the ability to move data between the Stack Pointer and Data Counters, or between the Stack Pointer and Accumulators is useful, since this allows a program to have more than one Stack. DC2 could be used as a buffer for the Stack Pointer, for example; now, by exchanging the contents of DC2 and SP, two Stacks could be accessed.

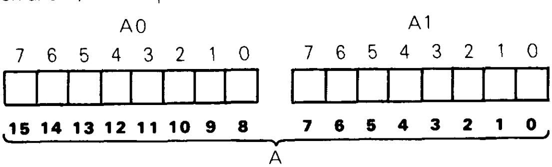
COMPUTED JUMP

Moving data between the Accumulators and the Program Counter allows program logic to compute jump addresses. This is very useful in branch tables, which are illustrated later in this chapter.

We will therefore provide Data Move and Data Exchange instructions as follows:



In the description above, X = A specifies a 16-bit value formed out of the two Accumulators as follows:



This instruction: **MOVE**

MOV S,D

Will move the register contents specified by S to the register specified by D. S and D must be one of the eight valid pairs shown; therefore, these Moves are legal:

MOV	A1,A0	Move A1 contents to A0
MOV	A0,A1	Move A0 contents to A1
MOV	SP,DC1	Move Stack Pointer contents to DC1

This Move is illegal:

MOV DC1,DC0

but the intended operation could be achieved via these two legal Moves:

MOV	DC1,A	Move DC1 contents to Accumulators
MOV	A,DC0	Move Accumulators to DC0

Recall the switch change test program; it used a register-to-register Move instruction as follows:

IN	4	Input new switch settings
XRA	SWITCH	Identify changed switches
MOV	A0,A1	Save A0 contents in A1
ANA	SWITCH	Identify switches that turned on

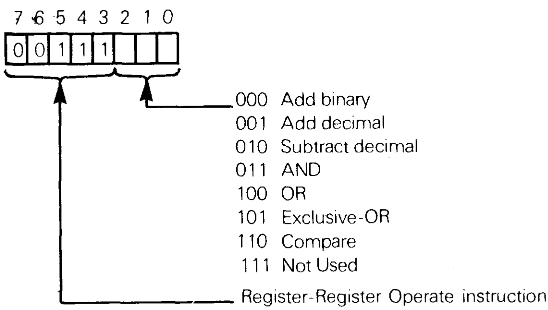
Exchange instruction mnemonics will be: **EXCHANGE**

X S,D

The same rules apply to S and D as described for MOV.

REGISTER-REGISTER OPERATE INSTRUCTIONS

Because our microcomputer has a number of secondary memory reference instructions, it needs very few Register-Register Operate instructions; the following seven instructions, which parallel the secondary memory reference instructions, will do:



The four status flags are set or reset to reflect the results of the operation.

Because there is an A0 - A1 Exchange instruction, we have only one set of Register-Register Operate instructions, where A0 is always the destination of the result.

The Register-Register Operate instructions will use these mnemonics:

ADD BINARY
ADD DECIMAL
SUBTRACT DECIMAL
AND
OR
EXCLUSIVE OR
COMPARE

AB	Add A1 to A0 binary
AD	Add A1 to A0 decimal
SD	Subtract A1 from A0 decimal
AND	AND A1 with A0
OR	OR A1 with A0
XOR	XOR A1.with A0
CMP	Compare A1 with A0

None of the Register-Register Operate instructions have operands.

These three instructions will allow A1 to be the destination of any Register-Register Operate instruction:

X	A0,A1	Exchange A0 and A1 contents
AB		Add binary; the result is in A0
X	A0,A1	Exchange A0 and A1 contents

REGISTER-REGISTER OPERATE INSTRUCTIONS JUSTIFICATION

Register-Register Operate instructions are convenient to have, but not vital, since they do nothing that could not be done using Load, Store and secondary memory reference instructions.

Register-Register Operate instructions will execute faster than equivalent secondary memory reference instructions, since secondary memory reference instructions require one data byte to be fetched from memory — and that takes time.

ACCUMULATOR DATA COUNTER ADDITION

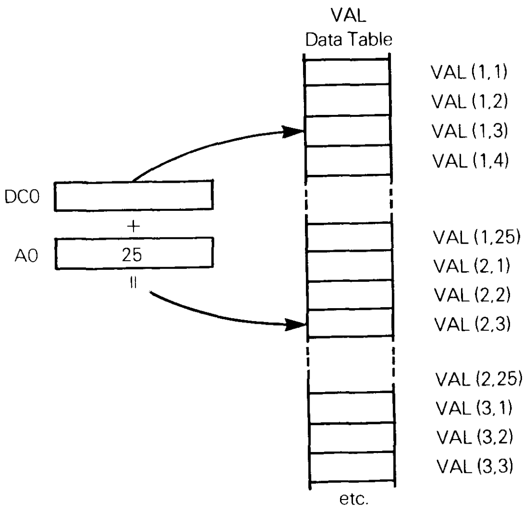
There is one further set of Register-Register Operate instructions which will prove very useful; we will allow the contents of Accumulator A0 to be added, as a signed binary number, to any one of the Data Counters. This allows a data address displacement to be computed, then added to (or subtracted from) a Data Counter.

ADDRESSING MATRICES

This instruction is particularly useful in matrix arithmetic, where doubly subscripted parameters such as

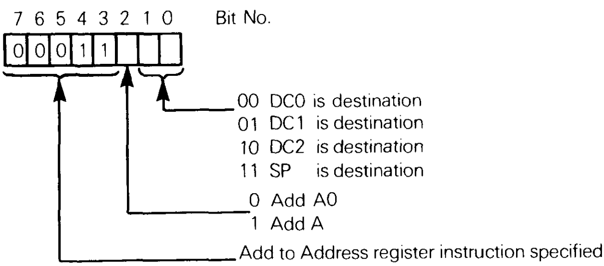
VAL (X,Y)

may be used. If the dimension of Y is known, each increment of X may be handled by adding the dimension of Y to the Data Counter which is addressing VAL. This is illustrated as follows:

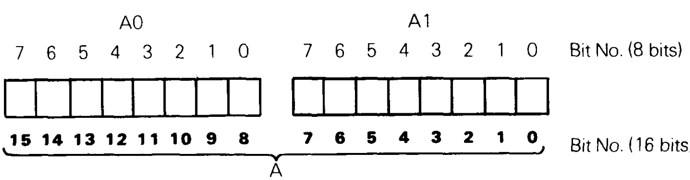


To extend this type of matrix handling, **we will also allow A0 and A1, treated as a 16-bit unit, to be added to any Data Counter.**

We now have these instruction codes:



A specifies the 16-bit unit:



We will use these mnemonics:

DAD S,D

S is the source, and may be A0 or A; no other options are allowed.

D is the destination, and may be DC0, DC1, DC2 or SP; no other options are allowed.

The Accumulator-Data Counter Addition instruction is also useful for creating branch tables.

BRANCH TABLES

A branch table is a list of addresses, identifying a number of programs, just one of which must be executed, based on current program logic.

First we will create a table of program starting addresses; because this is not a simple concept, we will illustrate it with an example that uses real, but arbitrary numbers:

```
ADDR1   EQU   H'1247'   Start of Program 1
ADDR2   EQU   H'183C'   Start of Program 2
ADDR3   EQU   H'28CA'   Start of Program 3
        etc.
        ORG   H'0800'
BTBL    DA    ADDR1
        DA    ADDR2
        DA    ADDR3
        etc.
```

EQUATE DIRECTIVE

The EQU mnemonic, recall, is an Assembler directive; it tells the Assembler what values to assign to the symbols ADDR1, ADDR2, ADDR3, etc.

DEFINE ADDRESS DIRECTIVE

The DA mnemonic is a "Define Address" Assembler directive; it tells the Assembler to place the 16-bit value provided by the operand in the next two currently identified memory locations.

ORIGIN DIRECTIVE

The ORG mnemonic is an Assembler directive which provides the current memory address. In this case, it defines the current memory address as 0800_{16}; in terms of a memory map, the above instructions result in these six data bytes:

	Memory Address	Memory	
	07FF		
BTBL ⟶	0800	12	ADDR1
	0801	47	
	0802	18	ADDR2
	0803	3C	
	0804	28	ADDR3
	0805	CA	
	0806		

The label BTBL, note, becomes a symbol with the value 0800_{16}.

Now suppose a program number is in Accumulator A0; we can execute the program identified by the program number as follows:

```
LIM   DC0,BTBL   Load the beginning address for program addresses into DC0
DAD   A0,DC0     Add the table number twice,
DAD   A0,DC0     since each address occupies two bytes
LNA   DC0        Load the address identified by DC0
LMB   DC0
MOV   A,PC       Move this address to PC
```

Look at what happens:

1) The LIM instruction loads 0800_{16} into DC0.
2) Suppose Accumulator A0 contains 2; the two DAD instructions add 4 to DC0, which now contains 0804_{16}.
3) The LNA instruction loads the contents of memory location 0804_{16} into A0, then increments DC0. Now A0 contains 28_{16} and DC0 contains 0805_{16}.
4) The LMB instruction loads the contents of memory location 0805_{16} into A1. Now A1 contains CA_{16}.
5) The MOV instruction moves the value $28CA_{16}$ into the Program Counter, forcing a jump to memory location $28CA_{16}$.

When would you use a branch table? One example is given in the description of interrupt instructions.

REGISTER OPERATE INSTRUCTIONS

Register Operate instructions modify the contents of a single register; no other register's contents are modified in any way.

Some Register Operate instructions are absolutely necessary, whereas others are nothing more than conveniences. We will therefore identify the ways in which a register's contents may be modified, and determine whether the operation is necessary, or just a convenience.

INCREMENT AND DECREMENT

The need to increment and decrement registers' contents is universal; whenever a register contains a counter or index, there is the probability that it will have to be incremented or decremented. To some extent, the auto increment and auto decrement variations of implied addressing makes the need to increment and decrement the Data Counters less vital; still it is a useful capability, since it allows addresses to be incremented or decremented selectively — not always.

COMPLEMENT

Since we have no binary subtract instructions, it is vital that there be an instruction to complement at least one of the Accumulators. Complementing the Data Counters serves no useful purpose. See Chapter 2 for a discussion of twos complement subtraction.

CLEAR REGISTER

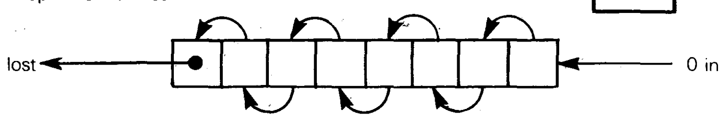
It must be possible to zero each Accumulator; this is a frequent prerequisite before performing addition, or simply as an initialization step. Zeroing the Address Registers is not necessary, since they have data loaded into them as the most frequent operation.

SHIFT AND ROTATE

Shift and rotate operations are very important for two reasons: they are vital to most multiplication and division algorithms, and they are frequently used in counting operations.

SHIFT

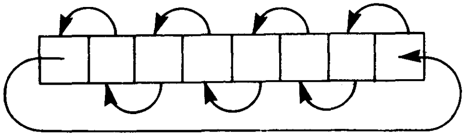
A shift operation is linear:



Thus a simple shift left, as illustrated above, will move each bit to the next bit to the left; the high order bit having no bit to the left, will be lost. There being no bit to the right of the low order bit, 0 will be moved into the low order bit.

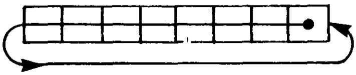
ROTATE

A rotate operation is circular; the high and low order bits would be assumed adjacent:

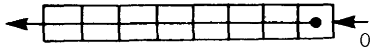


SIMPLE SHIFT AND ROTATE

Numerous variations of shift and rotate operations are possible; you can shift or rotate left:

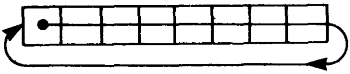


Simple Rotate Left



Simple Shift Left

You can rotate or shift right:



Simple Rotate Right



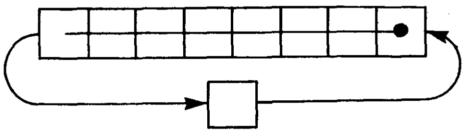
Simple Shift Right

A right rotate (or shift) is equivalent to dividing by 2, while the left rotate (or shift) is equivalent to multiplying by 2, and can be reproduced by adding the contents of a register to itself.

SHIFT AND ROTATE THROUGH CARRY

A shift or rotate may occur through the Carry status, in which case the shift and rotate become identical operations:

high order bit becomes new value of Carry

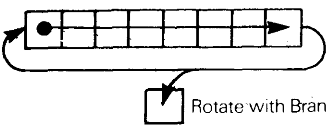


old value of Carry to low order bit

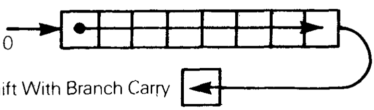
A left shift/rotate through Carry is illustrated above; the equivalent right shift through Carry is self-evident.

SHIFT AND ROTATE WITH BRANCH CARRY

Another variation branches a bit into the Carry status, but excludes the old Carry status from the shift or rotate:



Rotate with Branch Carry

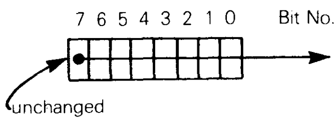


Shift With Branch Carry

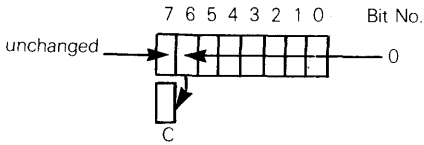
The shift with branch-carry is very useful as the first in a multibyte shift operation, where an initial value of 0 must be assumed for the Carry status.

ARITHMETIC SHIFT

The shift may also be arithmetic and propagate the high order bit (sign bit) to the right:



An arithmetic shift left will maintain the sign bit, and shii. out of the penultimate bit into the Carry:



SHIFTING BINARY CODED DECIMAL DATA

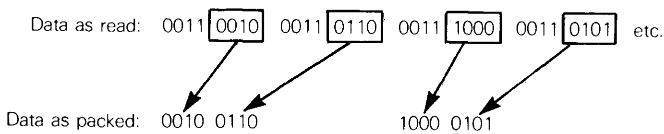
A four-bit shift, left or right, is very useful in microcomputer applictions, which frequently process numeric data.

As was discussed in Chapter 2, binary-coded decimal digits each occupy four bits; each byte holds two BCD digits. A four-bit shift is therefore equivalent to a single decimal digit shift, left or right; that is, it is equivalent to multiplying or dividing by ten.

The four-bit left and right shift also makes it easy to pack and unpack ASCII characters. Recall that the ASCII representation of a decimal digit appears as follows:

0 = 00110000
1 = 00110001
2 = 00110010
3 = 00110011
4 = 00110100
5 = 00110101
6 = 00110110
7 = 00110111
8 = 00111000
9 = 00111001

Suppose a string of ASCII digits are being read through an I/O port and must be packed in BCD format, two digits per byte as follows:



The four-bit shift is a natural for this operation; we will settle on some shift mnemonics, then write a program to perform this BCD packing operation.

How many, and which shift/rotate instructions should we have?

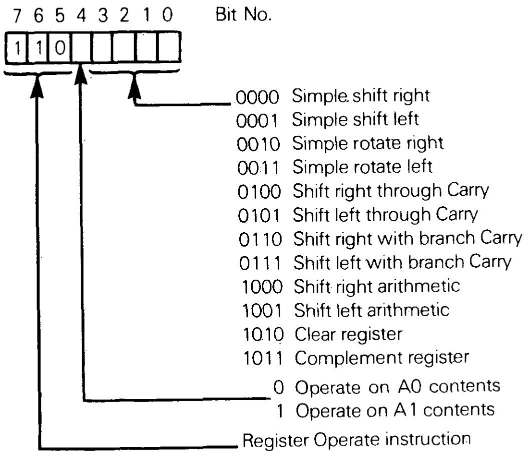
Shift and rotate instructions are usually inadequately rpresented in microcomputer instruction sets. We will have such instructions for the two Accumulators only, but we will provide shifts and rotates, without Carry (simple), with Carry, and with branched Carry.

Shifting Data Counter contents would not be very useful; it would provide a 16-bit shift, but that is a luxury we will have to forego.

We will include two versions of the four-bit left and right shift. One will operate on the contents of either Accumulator A0 or A1; the other will operate on the combined unit as a 16-bit number. In each case, since we are dealing with four-bit units the Accumulator will be ignored during the shift operation.

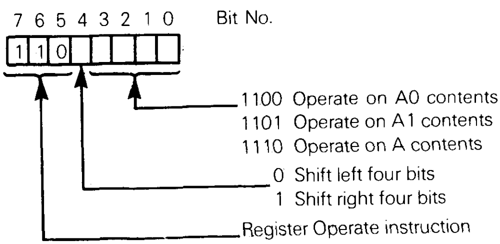
We can now summarize the Register Operate instruction object codes as follows.

For the operations which are confined to the Accumulators, these are the instructions and their object codes:

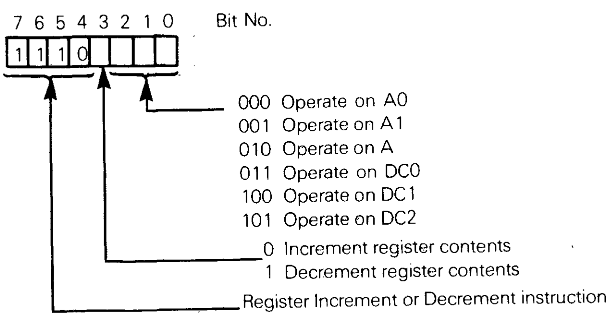


The shift and rotate instructions may modify the Carry status. The Complement instruction will affect the Zero status. No other status flags will be changed.

The two 4-bit shift instructions can operate on A0, A1, A(A0 - AI 16-bit unit). Object codes for these instructions will be as follows:



The Increment and Decrement instructions operate on the Accumulators and on the Address registers; they will use these object codes:



The Shift and Rotate instructions will have these mnemonics, without operands:

SHIFT AND ROTATE INSTRUCTIONS

SHRA	SHIFT A0 CONTENTS RIGHT SIMPLE
SHRB	SHIFT A1 AI CONTENTS RIGHT SIMPLE
SHLA	SHIFT A0 CONTENTS LEFT SIMPLE
SHLB	SHIFT A1 CONTENTS LEFT SIMPLE
RORA	ROTATE A0 RIGHT SIMPLE
RORB	ROTATE A1 RIGHT SIMPLE
ROLA	ROTATE A0 LEFT SIMPLE
ROLB	ROTATE A1 LEFT SIMPLE
SRCA	SHIFT A0 RIGHT THROUGH CARRY
SRCB	SHIFT A1 RIGHT THROUGH CARRY
SLCA	SHIFT A0 LEFT THROUGH CARRY
SLCB	SHIFT A1 LEFT THROUGH CARRY
SRBA	SHIFT A0 RIGHT WITH BRANCH CARRY
SRBB	SHIFT A1 RIGHT WITH BRANCH CARRY
SLBA	SHIFT A0 LEFT WITH BRANCH CARRY
SLBB	SHIFT A1 LEFT WITH BRANCH CARRY
SRAA	SHIFT A0 RIGHT ARITHMETIC
SRAB	SHIFT A1 RIGHT ARITHMETIC
SLAA	SHIFT A0 LEFT ARITHMETIC
SLAB	SHIFT A1 LEFT ARITHMETIC
SR4A	SHIFT A0 RIGHT FOUR BITS
SR4B	SHIFT A1 RIGHT FOUR BITS
SL4A	SHIFT A0 LEFT FOUR BITS
SL4B	SHIFT A1 LEFT FOUR BITS
SR4	SHIFT A0 AND A1 RIGHT FOUR BITS
SL4	SHIFT A0 AND A1 LEFT FOUR BITS

These are the mnemonics we will use for Register Operate instructions:

INCREMENT REGISTER

INC R

This specifies the "Increment Register" instruction; R may be A0, A1, A, DC0, DC1 or DC2.

The "Decrement Register" instruction will differ only in the mnemonic, as follows:

DECREMENT REGISTER

DEC R

Complement and Clear will apply to the two Accumulators only, and will have these mnemonics:

COMPLEMENT

CLEAR

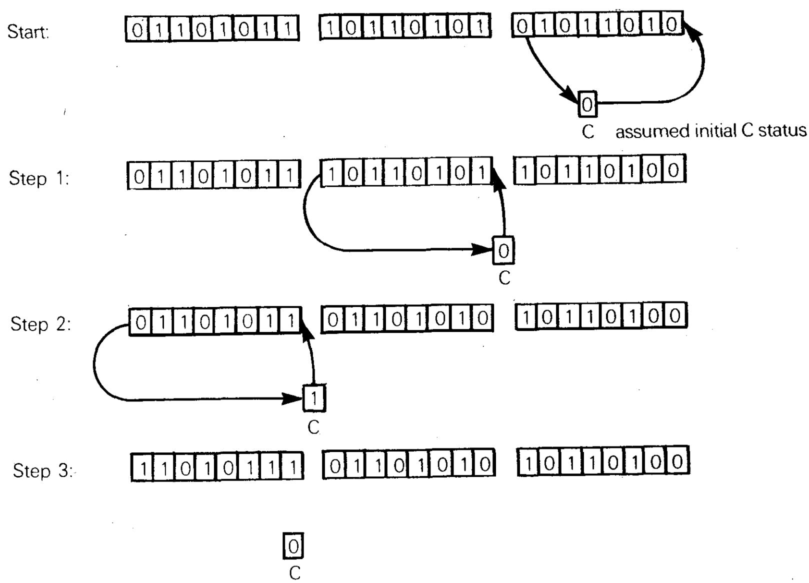
CLA	Clear A0
CLB	Clear A1
COA	Complement A0
COB	Complement A1

These four instructions have no operand.

We will now illustrate the value of Register Operate instructions with some examples of how these instructions may be used.

SHIFT MULTIBYTE

Consider multibyte shifts; they allow multibyte numbers to be multiplied and divided. A rotate through Carry will propagate a shift down a number of bytes, since the high order bit of each byte propagates into the Carry status, then into the low order bit of the next byte. This may be illustrated for the following simple, three-byte left shift:



The program to perform this operation is as follows:

LIM	DC0,BUFA	Load the buffer starting address in DC0
LMA	DC0	Load low order byte into A0 via DC0
SLBA		Shift left with branch carry
SDA	DC0	Store the result back; decrement DC0
LMA	DC0	Load the second byte into AC0
SLCA		Shift left with carry
SDA	DC0	Store result back; decrement DC0
LMA	DC0	Load last byte
SLCA		Shift left with carry
SMA	DC0	Store result back

The LIM instruction simply loads the address of the last byte into Data Counter DC0.

The next three instructions, LMA, SLBA and SDA accomplish step 1. First, LMA loads the low order byte into Accumulator A0 but does not modify the address in DC0, since we will want to return the shifted result to the same address. The SLBA instruction is very useful at this point, because it performs a branch carry; we do not know what the Carry status is before entering this routine, but with the SLBA instruction we do not care; this instruction loads 0 into the low order bit of A0, and it moves the high order bit of A0 into the Carry status, ready to be shifted into the next byte. The SDA instruction stores the shifted contents of A0 back into the memory byte from which the unshifted source came; then the address in DC0 is decremented to point to the second byte.

The next three instructions, LMA, SLCA and SDA perform step 2. These three instructions differ from the previous three instructions only in that a shift left with carry must now be performed since the Carry status represents the high order bit of the previous byte, which must become the low order bit of the current byte.

Step 3 is accomplished via the last three instructions, LMA, SLCA and SMA; these three instructions differ from the three step 2 instructions only in that we do not bother to decrement the address in DC0, since there are no more bytes to be shifted.

Observe that since only three bytes are to be shifted, we do not use an instruction loop. The whole of the program above only occupies 12 bytes, three for the Load Immediate into DC0 instruction, one each for the remaining instructions. We could condense the three steps into one set of three repeated instructions so long as we can change the SLBA to a SLCA instruction, and so long as the final SMA instruction becomes an SDA instruction. The program now appears as follows:

```
        Clear Carry status which must initially be 0
        LIM     DC0,BUFA     Load buffer starting address in DC0
        LIM     A1,3         Load byte count into A1
LOOP    LMA     DC0          Load next byte into A0, via DC0
        SLCA                 Shift left with Carry
        SDA     DC0          Store the result back; decrement DC0
        DEC     A1           Decrement byte count
        BNZ     LOOP         Return if not end
```

We now have a program with eight instructions versus the previous ten. But these eight instructions are still going to occupy 12 bytes; three for the Load Immediate into DC0, two each for the Load Immediate into A1 and the Branch on Non-Zero, and one each for the remainder. This is another example of the fact that when a loop has very few iterations, a branch-and-loop program structure offers few economies as compared to a once-through program structure.

SWITCH TESTING

Next consider switch testing. The eight switches we described when justifying secondary memory reference instructions could be tested for "on" or "off" status in a program loop as follows:

1) Load 00000001 into A1. We are going to use A1 as a switch counter. Its contents will be shifted left with branch carry until a 1 appears in the Carry status, which will indicate that eight shifts have been performed.
2) Load switch settings into A0.
3) Shift A0 one bit right with branch carry. The low order bit of A0 is now in the Carry status.
4) Save A0 and A1 in DC2. The Carry status still reflects the low order bit of A0, since a Move instruction will not affect the status flags.
5) Branch on "carry true" to "switch on" program. Otherwise continue with "switch off" program.

6) When the "switch on" or "switch off" program has completed execution, reload A0 and A1 from DC2.

7) Shift A1 left one bit with branch carry. If Carry is set, we are done. If Carry is not set, return to step 3 above.

The program steps required to implement the above logic are as follows:

```
         LIM    A1,1    Load 01 into A1
         IN     4       Input switch settings from I/O port 4
LOOP     SRBA           Shift A0 right with branch carry
         MOV    A,DC2   Save A0 and A1 in DC2
         BC     SWON    Branch on C=1 to "switch on" program
```

"Switch off" program logic appears here.

```
MOV    DC2,A    Restore A0 and A1 from DC2
SLBB            Shift A1 left with branch carry
```

PACKING ASCII DIGITS

Now consider the program steps needed to pack the numeric bits (low order four bits) of ASCII numeric digit representations; two numeric digits will be packed per byte, as described directly before shift object codes illustration.

These steps would be required to pack digits:

Step 1 — read in one ASCII digit and store in Accumulator A0.

Step 2 — shift left four bits.

Step 3 — move the contents of A0 to A1. A1 now contains the high order digit as follows:

ASCII digit: 0011XXXX

After four bit shift left: XXXX0000

Step 4 — Input the next ASCII digit to Accumulator A0.

Step 5 — mask out the high order four bits of A0. A0 now contains the low order digit as follows:

ASCII digit: 0011YYYY

After masking high order bits: 0000YYYY

Step 6 — Add A1 to A0. A0 now contains the high and low order digits as follows:

0000YYYY + XXXX0000 = XXXXYYYY

Step 7 — store the two packed digits in memory (we will assume the correct buffer is addressed by DC1).

Step 8 — return to step 1 for the next two ASCII digits.

We will assume that ASCII digits are input at I/O Port 5, and bit 0 of I/O Port 6 is set to 1 by the inputting device whenever it has transmitted an ASCII digit to I/O Port 5.

Program steps are as follows:

```
LOOP1   IN      6       Input status
        SRBA            Shift bit 0 of A0 with branch carry
        BNC     LOOP1   If Carry is 0, input status again
        OUT     6       If Carry is 1, output A0 to I/O Port 6
                        This clears the status
        IN      5       Input the next ASCII digit
        SL4A            Shift left 4 bits
        MOV     A0,A1   Save in A1
LOOP2   IN      6       Repeat first five instructions
        SRBA            to input next ASCII digit
        BNC     LOOP2
        OUT     6
        IN      5
        NIA     H'0F'   Mask out high order four bits
        AB              Add A1 to A0
        LDA     DC0     Store the two packed digits
        JMP     LOOP1   Return for next two digits
```

STACK INSTRUCTIONS

Since our microcomputer has a Stack, it must have Push instructions to move registers' contents onto the Stack; it must also have Pop instructions to move data off the Stack, and into registers.

JUMP-TO-SUBROUTINE

Many microcomputer manuals list the Jump-to-Subroutine instruction as a Stack instruction, since it pushes the Program Counter contents onto the Stack before loading a new address into the Program Counter.

PUSH

Push instructions will be used primarily for interrupt processing; programming examples are given along with interrupt handling instructions.

POP

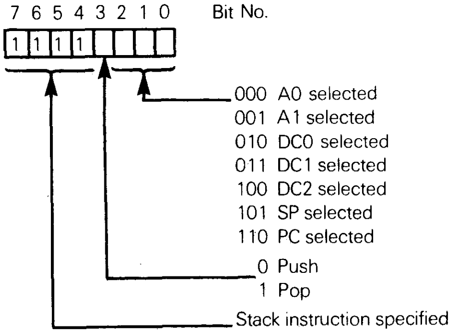
RETURN-FROM-SUBROUTINE

Pop instructions are used in interrupt processing, and in order to return from a subroutine; examples of the latter use are given shortly.

SUBROUTINE PARAMETER PASSING

Push and Pop instructions are sometimes used to pass data (parameters) to subroutines; we will illustrate this use of Push and Pop instructions later.

Our microcomputer will have Push and Pop instructions that reference the two Accumulators, and the four Address registers; object codes will be as follows:



The Push and Pop instructions will use this instruction format:

OP R

OP represents the instruction mnemonic; it will be PUSH or POP, for a Push or Pop instruction, respectively.

R will specify the register whose contents is to be pushed onto the Stack, or which is to receive data popped from the Stack. R may be A0, A1, DC0, DC1, DC2 or PC. No other symbol is allowed.

RETURN INSTRUCTION

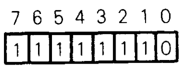
We will allow an additional instruction mnemonic for subroutine returns. The instruction:

POP PC

will move the two bytes at the top of the Stack into the Program Counter, thus effecting a return from the subroutine. The mnemonic:

RET

will perform the same operation, and generate the same object code; in other words, the RET mnemonic will generate the one object code byte:



As an example of Stack instructions' use, return to the data movement subroutine which was described along with Immediate instructions; the subroutine was listed like this:

```
MOVE    LIM     DC0,BUFA     Load source initial address
        LIM     DC1,BUFB     Load destination initial address
LOOP    LNA     DC0          Move data from source
        SSA     DC1,LOOP     to destination
        Return from Subroutine
```

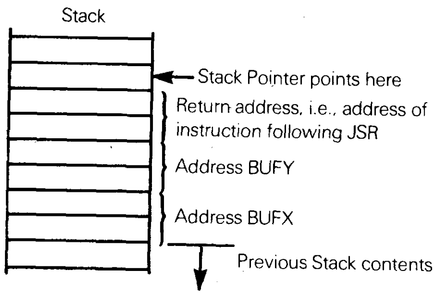
PARAMETER PASSING

In addition to adding a Return instruction, this subroutine can be made more useful if the beginning addresses for the source and destination buffers (BUFA and BUFB) are variable. Stack instructions provide one way (but not the best way) of making this possible.

Before calling subroutine MOVE, a program can push its version of the addresses BUFA and BUFB onto the Stack, as follows:

```
LIM     DC0,BUFX
PUSH    DC0
LIM     DC0,BUFY
PUSH    DC0
JSR     MOVE
```

The top of the Stack now looks like this:



Subroutine MOVE must be modified as follows:

```
MOVE    POP     DC2         Save the return address in DC2
        POP     DC1         Load BUFY as destination initial address
        POP     DC0         Load BUFX as source initial address
        PUSH    DC2         Replace return address at top of stack
LOOP    LNA     DC0         Move data from source
        SSA     DC1,LOOP    to destination
        RET                 Pop return address from stack
```

PARAMETER PASSING INSTRUCTIONS

Because subroutines are so frequently used, it is worth taking a look at instructions which make subroutines easier to use.

Let us return again to the data move subroutine which we have been developing up to this point.

In the first place, this subroutine simply moved data from a source buffer with a dedicated address, to a destination buffer with another dedicated address.

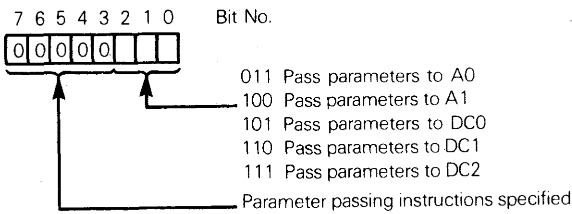
SUBROUTINE PARAMETERS

Next, when describing the Push and Pop instructions we improved on the versatility of this subroutine by allowing the calling program to specify the source and destination buffer beginning addresses. These two addresses are called "parameters", which the calling program passes to the subroutine. Parameter passing is a very important feature of subroutine handling; by making parameter passing easy, a microcomputer becomes a significantly more powerful device.

Parameter passing instructions are, in fact, quite simple to specify. What we will do is to allow parameters to follow the Jump-to-Subroutine instruction, then we will provide the microcomputer with a form of indirect addressing, where the two bytes at the top of the Stack become the memory address from which data will be fetched.

But before we explain this concept with pictures and examples, let us define the parameter passing instructions which our microcomputer will include.

First, there are the object codes that are to be used:



PASS PARAMETER INSTRUCTION

The instruction format for the Pass Parameter instruction will be as follows:

SPP R

SPP is the instruction mnemonic, and R identifies one of the registers A0, A1, DC0, DC1 or DC2; no other symbol is allowed for R.

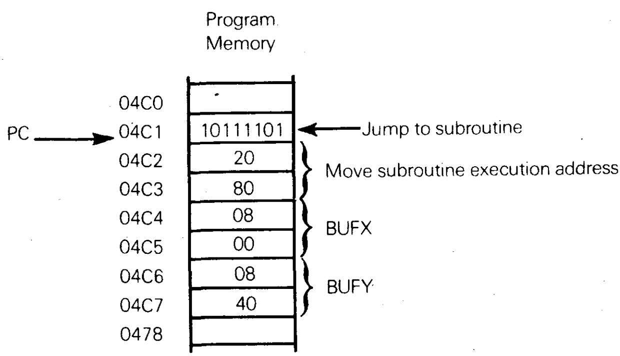
PASSING PARAMETERS TO SUBROUTINES

We will now develop a very efficient implementation of the data moving subroutine.

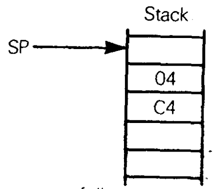
The subroutine will be called as follows:

```
JSR     MOVE    Call data move subroutine
DA      BUFX    Specify beginning source address
DA      BUFY    Specify beginning destination address
```

Recall that the DA mnemonic represents the Define Address Assembler directive. Suppose these instructions reside in memory as follows:



After the JSR instruction has executed, PC will contain 2080_{16}, which is the execution address for subroutine MOVE. The previous value of PC, $04C4_{16}$, will be at the top of the Stack:



The MOVE subroutine appears as follows:

MOVE	SPP	DC0	Load source starting address into DC0
	SPP	DC1	Load destination starting address into DC1
LOOP	LNA	DC0	Move data from source
	SSA	DC1,LOOP	to destination
	RET		Pop return address from Stack

The first SPP instruction causes the CPU to execute the following logic:

1) The two bytes at the top of the Stack are fetched into the CPU.
2) These two bytes are treated as a memory address. The contents of the memory location identified by this memory address are loaded into the high order byte of DC0. The memory address is then incremented. The memory address was $04C4_{16}$ and memory location $04C4_{16}$ contains 08_{16}. Therefore, at the end of this step, the high order byte of DC0 contains the value 08_{16} and the memory address has been incremented to $04C5_{16}$.
3) Step 2 is repeated, with the data fetched from memory going to the low order byte of DC0. At the end of this step DC0 contained 0800_{16}, and the memory address is now $04C6_{16}$.
4) Instruction execution is complete so the memory address is returned to the top of the Stack, which now holds $04C6_{16}$, not $04C4_{16}$.

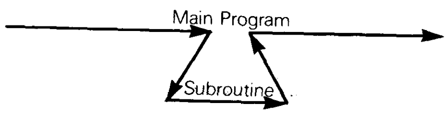
The second SPP instruction is a repeat of the first SPP instruction, except that DC1 is specified as the destination; therefore, at the conclusion of the SPP instruction, 0840_{16} will be stored in DC1, and the top two bytes of the Stack will hold the value $04C8_{16}$. This is the address of the next instruction to be executed following the two parameters, BUFX and BUFY. At the conclusion of the Move subroutine, the RET instruction will pop the value $04C8_{16}$ back into the Program Counter, thus allowing normal program execution to continue.

INTERRUPT INSTRUCTIONS

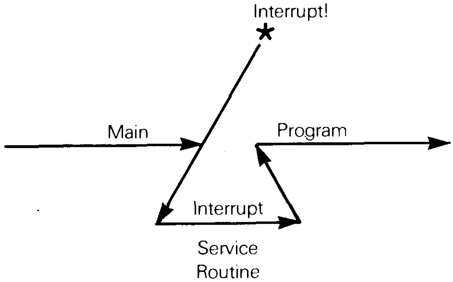
In reality we are going to talk about more than interrupt instructions. There are only three interrupt instructions; one disables all interrupts, the second enables all interrupts, and the third is a Return-from-Interrupt instruction.

How is our microcomputer going to handle interrupts?

There are many similarities between processing an interrupt and entering a subroutine; in each case, program execution temporarily branches from a main program to a secondary logic sequence, at the conclusion of which program execution returns to the main program. The difference between a subroutine and an interrupt is that a Jump-to-Subroutine is part of the scheduled mainstream logic:



An interrupt, on the other hand, is an unscheduled event, and the main program has no way of knowing when the interrupt will occur:



We discussed at some length, in Chapter 5, the various ways in which external devices can interrupt the CPU. Recall that as the CPU's interrupt protocol becomes more minicomputer-like, and more sophisticated, so also the cost and complexity of the external logic needed to meet the requirements of CPU interrupt protocol goes up. We will therefore adopt a very simple scheme. Interrupting devices will be daisy-chained on a single interrupt request line, and when the CPU sends out an acknowledge signal, the interrupting device will output a single byte of data to an I/O port with address FF_{16}. The CPU will interpret the data in I/O port FF_{16} as identifying the interrupting device.

As soon as the CPU acknowledges an interrupt, it will automatically do three things:

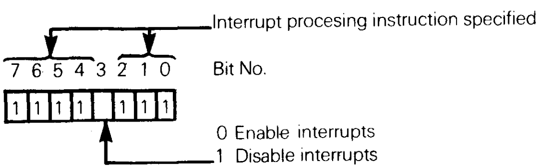
First, it will disable interrupts, thus preventing another interrupt from being processed before the current one has been adequately handled. An Enable Interrupt instruction must be executed by the program before any further interrupts can be handled.

Next the CPU will save the status flags' contents by pushing them onto the Stack.

Finally the CPU will push the Program Counter contents to the top of the Stack, and clear the Program Counter. This causes program execution to continue at memory location 0.

A Disable Interrupt instruction can be executed at any time to prevent any interrupts from being acknowledged; this condition will last until the Enable Interrupt instruction is re-executed.

Let us first look at the object code for the Enable and Disable Interrupt instructions:



The mnemonics for the two interrupt instructions will be:

ENABLE INTERRUPT

DISABLE INTERRUPT

DI

for Disable Interrupts, and

EI

for Enable Interrupts.

The Return-from-Interrupt instruction will do three things:

RETURN FROM INTERRUPT

First it will return the status flags, which were saved on the Stack automatically when the interrupt was acknowledged.

Then it will pop the return address from the Stack to the Program Counter.

Finally the Return-from-Interrupt instruction enables interrupts.

The Return-from-Interrupt instruction's object code will be:

7	6	5	4	3	2	1	0	Bit No.
0	0	1	1	1	1	1	1	

Bits 7–0: Return from interrupt

The instruction mnemonic will be:

RTI

To illustrate the use of interrupt instructions, we will show the program steps which follow an Interrupt Acknowledge.

We will also show the program steps which must be present at the end of the interrupt service routine.

Following an interrupt these steps must occur:

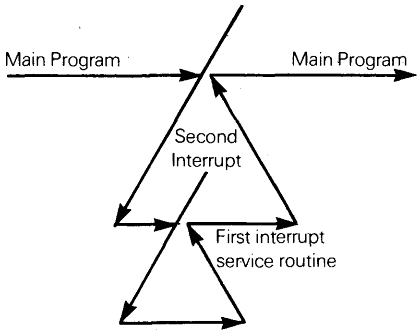
INTERRUPT ACKNOWLEDGE

1) At the time of the Interrupt Acknowledge, the CPU logic saves the status flags at the top of the Stack, pushes the Program Counter contents onto the top of the Stack, then disables interrupts. The Program Counter is zeroed, which means that program execution jumps to memory location 0.

2) Starting at memory location 0, there is a short program sequence which saves the contents of all CPU registers by pushing registers' contents onto the Stack. This is necessary, since the registers may be used in any way by the program which is about to be executed.

3) After all registers' contents have been saved on the Stack, the contents of I/O Port FF_{16} is read, and is used to compute the starting address of the particular program which will service the identified interrupting device.

4) The program which gets executed following step 3 may optionally contain an Enable Interrupt instruction. If this instruction is present, another interrupt may be processed, before the current interrupt has completed execution, as follows:



If interrupts are not enabled, then no further interrupts can be processed until their Return-from-Interrupt instruction is executed.

This is the instruction sequence which, given our interrupt service logic, must be present beginning at memory location 0:

ORG	0	
PUSH	A0	Save all registers' contents
PUSH	A1	on the stack
PUSH	DC0	
PUSH	DC1	
PUSH	DC2	
IN	H'FF'	Input device ID from I/O Port FF
LIM	DC0,BTBL	Load jump table base address
SHLA		Shift A0 left, simple, to multiply by 2
DAD	A0,DC0	Add A0 to DC0
LNA	DC0	Load the interrupt service routine
LMB	DC0	starting address
MOV	A,PC	Move the address to PC

This is what the above short program does.

SAVING REGISTERS ON STACK

Recall that the ORG mnemonic specifies the current memory address for the Assembler. The ORG mnemonic above tells the Assembler to start creating object code beginning at memory location 0.

The five Push instructions save the contents of all registers on the Stack.

The IN instruction will receive a device ID at I/O Port FF. We are assuming that within the time taken for the CPU to execute the Push instruction, the interrupting device will have been able to place its ID number at I/O Port FF. This ID number will be in Accumulator A0.

BRANCH TABLE

The instructions from LIM to MOV constitute a branch table. Branch tables were described along with the DAD Register-Register Operate instruction. Notice in the branch table instruction sequence above that a Shift instruction has been used to multiply the contents of A0 by 2 before adding to DC0; in the previous example, the DAD instruction was executed twice to achieve the same end result.

The address computed by the branch table becomes the beginning address of the interrupt service routine, which will now be executed to service the specific device which requested an interrupt. Once the interrupt service routine has completed execution, it will call a subroutine that reverses the interrupt acknowledge steps as follows:

```
RINT    POP     DC2     Restore all registers' contents
        POP     DC1
        POP     DC0
        POP     A1
        POP     A0
        RTI
```

RESTORING REGISTERS FROM STACK

Observe that registers are popped from the Stack in the reverse order to which they were pushed, since the Stack is a last-in-first-out storage unit.

The final RTI instruction will restore the saved status (which is on the Stack following the interrupt acknowledge) to the four status flags, then will load back into the Program Counter the memory address which was saved at the time of the interrupt acknowledge.

If interrupts are still disabled, the RTI instruction will re-enable interrupts.

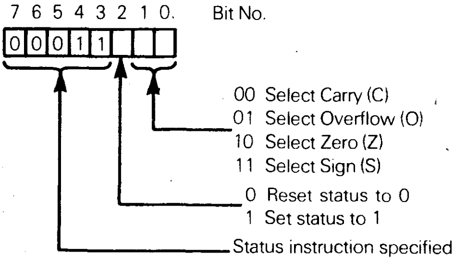
Program execution will now continue at the point where the interrupt occurred.

STATUS INSTRUCTIONS

Since we have four status flags, Sign (S), Carry (C), Overflow (O) and Zero (Z), it must be possible to set or reset these flags individually. The most common situation in which program logic will rquire a flag to be set is just before entering a program loop which contains a Branch on Condition instruction at the beginning of the loop. In the normal course of events, status flags will be set later in the loop to be tested when program logic comes back to the beginning of the loop. It must be possible to set status conditions before entering the loop, so that we can get by the Branch on Condition on the first pass.

There are also many multibyte arithmetic algorithms which require the Carry and Overflow statuses to be either cleared or set before starting the algorithm; subsequently, after each byte of the multibyte number is processed, carries are passed from one byte to the next via these two status flags, as described in Chapter 2.

We will therefore include these eight status instructions:



The Set Status instruction will have this format:

STATUS SET

SET X

The Reset Status instruction will have this format:

STATUS RESET

RES X

In each case, X may be C, O, Z or S, to identify one of the four status flags. No other symbol is allowed.

As an example of status instruction use, the multibyte, binary addition routine, described along with the secondary memory reference instructions, starts out by clearing the Carry status:

```
        RES   C          Clear Carry status
LOOP    LMA   DC0        Load next input byte
        ABA   DC1        Add binary from answer buffer
        SSA   DC1,LOOP   Store the result, increment and skip
```

Once in the loop, the binary addition instruction ABA sets and resets the Carry status appropriately.

HALT INSTRUCTION

Every microcomputer has a Halt instruction. When this instruction is executed, the microcomputer simply stops. In a minicomputer, or in a microcomputer that has a front panel, program execution is restarted by hitting a restart button on the panel. So far as the CPU is concerned, the reset signal which is input to the CPU (and was described in Chapter 4) must be pulsed in order to start execution after a Halt instruction.

In our microcomputer, and in many other microcomputers, the Halt instruction object code consists of all 0 bits. This is done with good reason, since unused memory words frequently contain all 0 bits. In the event that a program, while being debugged, makes a wild jump and tries to execute instructions in some area of memory where no instructions exist, there is a very good chance that it will pick up all 0s for the next instruction object code — which will cause the program to simply stop, and do as little harm as possible.

The Halt instruction mnemonic will be, appropriately:

HALT

AN INSTRUCTION SET SUMMARY

You will find that books describing individual microcomputers provide tables that summarize the microcomputer instruction set cryptically. These summary tables are very useful. Assuming that you have a general understanding of assembly languages, two or three pages tell you everything you need to know about operations performed when any instruction is executed.

We are going to summarize our hypothetical instruction set with summary Table 7-1. In Volume 2 similar tables will summarize the instruction sets for real microcomputers.

In Table 7-1, symbols are used as follows:

AC0	Accumulator AC0
AC1	Accumulator AC1
ADDR	A 16-bit memory address
C	Carry status
DATA	An 8-bit binary data unit
DC0	Data Counter DC0
DC1	Data Counter DC1
DC2	Data Counter DC2
DCX	Any data counter
DISP	An 8-bit signed binary address displacement
DST	Any destination register
I	Any status indicator
O	Overflow status
P	An I/O port number
PC	Program Counter
R	Any register
S	Sign status
SP	Stack Pointer
SRC	Any source register
SW	Statuses
Z	Zero Status
[ ]	Contents of location enclosed within brackets. If a register designation is enclosed within the brackets, then the designated register's contents are specified. If an I/O port number is enclosed within the brackets, then the I/O port contents are specified. If a memory address is enclosed within the brackets, then the contents of the addressed memory location are specified.
[[ ]]	Implied memory addressing; the contents of the memory location designated by the contents of a register.
$\wedge$	Logical AND
V	Logical OR
$\forall$	Logical Exclusive OR
$\leftarrow$	Data is transferred in the direction of the arrow.
$\longleftrightarrow$	Data is exchanged between the two locations designated on either side of the arrow.

Under the heading of STATUSES in Table 7-1, an X indicates statuses which are modified in the course of the instructions' execution. If there is no X, it means that the status maintains the value it had before the instruction was executed.

Table 7-1. A Summary Of The Hypothetical Microcomputer Instruction Set

TYPE	MNEMONIC	OPERAND(S)	BYTES	STATUSES C	O	Z	S	OPERATION PERFORMED
I/O	INS,IN	P	1,2					[AC0]←[P] Input to A0 from I/O Port P (only 0, 1 or 2 for INS)
	OUTS,OUT	P	1,2					[P]←[AC0] Output from A0 to I/O Port P (only 0, 1 or 2 for OUTS)
Primary Memory Reference	LRA,LRB	ADDR	3					[AC0]←[ADDR], [AC1]←[ADDR] Load to A0 or A1, use direct addressing
	SRA,SRB	ADDR	3					[ADDR]←[AC0], [ADDR]←[AC1] Output from A0 or A1, use direct addressing
	LMA,LMB	DCX	1					[AC0]←[[DCX]], [AC1]←[[DCX]] Load AC0 or AC1 using implied addressing
	LNA,LNB	DCX	1					[AC0]←[[DCX]], [AC1]←[[DCX]], and [DCX]←[DCX] + 1 Load AC0 or AC1 using implied addressing with auto increment
	LDA,LDB	DCX	1					[AC0]←[[DCX]], [AC1]←[[DCX]], and [DCX]←[DCX]-1 Load AC0 or AC1 using implied addressing with auto decrement
	SMA,SMB	DCX	1					[[DCX]]←[AC0], [[DCX]]←[AC1] Store AC0 or AC1 in memory using implied addressing
	SNA,SNB	DCX	1					[[DCX]]←[AC0], [[DCX]]←[AC1], and [DCX]←[DCX] + 1 Store AC0 or AC1 in memory using implied addressing with auto increment
	SDA,SDB	DCX	1					[[DCX]]←[AC0], [[DCX]]←[AC1], and [DCX]←[DCX]-1 Store AC0 or AC1 in memory using implied addressing with auto decrement
	LSA,LSB	DCX,ADDR	2					[AC0]←[[DCX]], [AC1]←[[DCX]] and [DCX]←[DCX] + 1 plus skip Load AC0 or AC1 using implied addressing with auto increment and skip
	SSA,SSB	DCX,ADDR	2					[[DCX]]←[AC0], [[DCX]]←[AC1] and [DCX]←[DCX] + 1 plus skip Store AC0 or AC1 in memory using implied addressing with auto increment and skip

Table 7-1. (Continued)

TYPE	MNEMONIC	OPERAND(S)	BYTES	STATUSES C	O	Z	S	OPERATION PERFORMED
Secondary Memory Reference	ABA,ABB	ADDR	3	X	X	X	X	[AC0]←[AC0] + [ADDR] + [C], [AC1]←[AC1] + [ADDR] + [C] Add binary with carry to A0 or A1 using direct addressing
	ABA,ABB	DCX	1	X	X	X	X	[AC0]←[AC0] + [[DCX]] + [C], [AC1]←[AC1] + [[DCX]] + [C] Add binary with carry to A0 or A1 using implied addressing
	ADA,ADB	ADDR	3	X	X	X	X	[AC0]←[AC0] + [ADDR] + [C], [AC1]←[AC1] + [ADDR] + [C] Add decimal with carry to A0 or A1 using direct addressing
	ADA,ADB	DCX	1	X	X		X	[AC0]←[AC0] + [[DCX]] + [C], [AC1]←[AC1] + [[DCX]] + [C] *Add decimal with carry to A0 or A1 using implied addressing*
	DSA,DSB	ADDR	3	X	X	X	X	[AC0]←[AC0] - [ADDR] - [C], [AC1]←[AC1] - [ADDR] - [C] Subtract decimal with borrow from A0 or A1 using direct addressing
	DSA,DSB	DCX	1	X	X	X	X	[AC0]←[AC0] - [[DCX]] - [C], [AC1]←[AC1] - [[DCX]] - [C] Subtract decimal with borrow from A0 or A1 using implied addressing
	ANA,ANB	ADDR	3			X		[AC0]←[AC0]∧[ADDR], [AC1]←[AC1]∧[ADDR] AND with A0 or A1 using direct addressing
	ANA,ANB	DCX	1			X		[AC0]←[AC0]∧[[DCX]], [AC1]←[AC1]∧[[DCX]] AND with A0 or A1 using implied addressing
	ORA,ORB	ADDR	3			X		[AC0]←[AC0]∨[ADDR], [AC1]←[AC1]∨[ADDR] *OR with A0 or A1 using direct addressing*
	ORA,ORB	DCX	1			X		[AC0]←[AC0]∨[[DCX]], [AC1]←[AC1]∨[[DCX]] OR with A0 or A1 using implied addressing
	XRA,XRB	ADDR	3			X		[AC0]←[AC0]∀[ADDR], [AC1]←[AC1]∀[ADDR] Exclusive OR with A0 or A1 using direct addressing
	XRA,XRB	DCX	1			X		[AC0]←[AC0]∀[[DCX]], [AC1]←[AC1]∀[[DCX]] Exclusive OR with A0 or A1 using implied addressing
	CMA,CMB	DCX	1	X	X	X	X	Compare memory with A0 or A1 using implied addressing
Immediate	LIM	R,DATA	2					[DST]←DATA Load immediate into R (R = A0 or A1)
	LIM	R,DATA	3					[DST]←DATA Load immediate into R (R = DC0, DC1, DC2, SP)

Table 7-1. (Continued)

TYPE	MNEMONIC	OPERAND(S)	BYTES	STATUSES C	O	Z	S	OPERATION PERFORMED
Immediate Operate	AIA,AIB	DATA	2	X	X	X	X	[AC0]←[AC0] + DATA + [C], [AC1]←[AC1] + DATA + [C] Add binary immediate to A0 or A1
	NIA,NIB	DATA	2			X		[AC0]←[AC0] Λ DATA, [AC1]←[AC1] Λ DATA AND immediate with A0 or A1
	OIA,OIB	DATA	2			X		[AC0]←[AC0] V DATA, [AC1]←[AC1] V DATA OR immediate with A0 or A1
	CIA, CIB	DATA	2	X	X	X	X	Compare immediate with A0 or A1
Jump	JMP	ADDR	3					[PC]←ADDR Jump to instruction with label ADDR
	JSR	ADDR	3					[[SP]]←[PC], [PC]←ADDR, [SP]←[SP] + 1 Jump to subroutine starting at ADDR
Branch On Condition	BE,BZ	DISP	2					If [Z] = 1, [PC]←[PC] + DISP Branch on Z = 1
	BNZ	DISP	2					If [Z] = 0, [PC]←[PC] + DISP Branch on Z = 0
	BGE,BC	DISP	2					If [C] = 1, [PC]←[PC] + DISP Branch on C = 1
	BNC,BL	DISP	2					If [C] = 0, [PC]←[PC] + DISP Branch on C = 0
	BO	DISP	2					If [O] = 1, [PC]←[PC] + DISP Branch on O = 1
	BNO	DISP	2					If [O] = 0, [PC]←[PC] + DISP Branch on O = 0
	BP	DISP	2					If [S] = 0, [PC]←[PC] + DISP Branch on S = 0
	BN	DISP	2					If [S] = 1, [PC]←[PC] + DISP Branch on S = 1

Table 7-1. (Continued)

TYPE	MNEMONIC	OPERAND(S)	BYTES	STATUSES C	O	Z	S	OPERATION PERFORMED
Reg-Reg Move	MOV	SRC,DST	1					[DST]←[SRC] Move contents of SRC to DST SRC=A0, A or SP. DST=A1, DC0, DC1, DC2 or PC.
	X	SRC,DST	1					[DST]←→[SRC] Exchange SRC and DST contents
Register-Register Operate	AB		1	X	X	X	X	[AC0]←[AC0] + [AC1] + [C] Add binary A1 to A0
	AD		1	X	X	X	X	[AC0]←[AC0] + [AC1] + [C] Add decimal A1 to A0
	SD		1	X	X	X	X	[AC0]←[AC0] - [AC1] - [C] Subtract decimal A1 from A0
	AND		1			X		[AC0]←[AC0] Λ [AC1] AND A1 with A0
	OR		1			X		[AC0]←[AC0] V [AC1] OR A1 with A0
	XOR		1			X		[AC0]←[AC0] ∀ [AC1] Exclusive OR A1 with A0
	CMP		1			X		Compare A1 with A0
	DAD	SRC,DST	1	X	X	X	X	[DST]←[DST] + [SRC] + [C] Add binary SRC to DST SRC=A0 or A. DST=DC0, DC1, DC2 or SP.
Register Operate	SHRA,SHRB		1					Shift A0 or A1 right, simple
	SHLA,SHLB		1					Shift A0 or A1 left simple
	RORA,RORB		1					Rotate A0 or A1 right, simple
	ROLA,ROLB		1					Rotate A0 or A1 left, simple
	SRCA,SRCB		1	X				Shift A0 or A1 right through carry
	SLCA,SLCB		1	X				Shift A0 or A1 left through carry
	SRBA,SRBB		1	X				Shift A0 or A1 right with branch carry
	SLBA,SLBB		1	X				Shift A0 or A1 left with branch carry

Table 7-1. (Continued)

TYPE	MNEMONIC	OPERAND(S)	BYTES	STATUSES C	O	Z	S	
Register Operate (continued)	SRAA,SRAB		1	X	X			Shift A0 or A1 right arithmetic
	SLAA,SLAB		1	X	X			Shift A0 or A1 left arithmetic
	SR4A,SR4B		1					Shift A0 or A1 right 4 bits
	SL4A,SL4B		1					Shift A0 or A1 left 4 bits
	SR4		1					Shift A0 and A1 right 4 bits
	SL4		1					Shift A0 and A1 left 4 bits
	INC	R	1				X	[R]←[R] + 1 Increment Register (R=A0, A1, A, DC0, DC1, DC2)
Stack	PUSH	R	1					[[SP]]←[R], [SP]←[SP] + 1 Push Register R contents onto Stack R=A0, A1, DC0, DC1, DC2, SP, PC
	POP	R	1					[R]←[[SP]], [SP]←[SP] -1 Pop top of Stack to Register R
	RET		1					[PC]←[[SP]], [SP]←[SP] -1 Same as POP PC
Param Pass	SPP	R	1					[R]←[[[SP]]], [[SP]]←[[SP]] + 1 Pass parameter to Register R R=A0, A1, DC0, DC1 or DC2
Interrupt	DI		1					Disable interrupts
	EI		1					Enable interrupts
	RTI		1					[SW]←[[SP]], [PC]←[[SP]] Return from interrupt
Status	SET	I	1	X	X	X	X	[I]←1 Set Status I to 1 (I = C, O, Z, S)
	RET	I	1	X	X	X	X	[I]←0 Reset status I to 0
	HALT		1					Halt

APPENDIX A
STANDARD CHARACTER CODES

Hexadecimal Representation	ASCII (7 bit)	EBCDIC (8 bit)
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
A		
B		
C		
D		
E		
F		
10		
11		
12		
13		
14		
15		
16		
17		
18		
19		
1A		
1B		
1C		
1D		
1E		
1F		
20	blank	
21	!	
22	"	
23	#	
24	$	
25	%	
26	&	
27	'	
28	(	
29	)	
2A	*	
2B	+	
2C	,	
2D	-	
2E	.	
2F	/	
30	0	
31	1	
32	2	
33	3	
34	4	
35	5	
36	6	
37	7	
38	8	
39	9	
3A	:	
3B	;	
3C	<	
3D	=	
3E	>	

Hexadecimal Representation	ASCII (7bit)	EBCDIC (8 bit)
3F	?	
40	@	blank
41	A	
42	B	
43	C	
44	D	
45	E	
46	F	
47	G	
48	H	
49	I	
4A	J	|
4B	K	.
4C	L	(
4D	M	(
4E	N	+
4F	O	!
50	P	&
51	Q	
52	R	
53	S	
54	T	
55	U	
56	V	
57	W	
58	X	
59	Y	
5A	Z	[
5B	[	$
5C	\	*
5D	]	)
5E		;
5F		∧
60		
61	a	
62	b	
63	c	
64	d	
65	e	
66	f	
67	g	
68	h	
69	i	
6A	j	
6B	k	,
6C	l	%
6D	m	-
6E	n	>
6F	o	?
70	p	
71	q	
72	r	
73	s	
74	t	
75	u	
76	v	
77	w	
78	x	
79	y	
7A	z	:
7B		#
7C		@
7D		'

APPENDIX A (continued)

Hexadecimal Representation	ASCII (7 bit)	EBCDIC (8 bit)
7E		=
7F		"
80		
81		a
82		b
83		c
84		d
85		e
86		f
87		g
88		h
89		i
8A		
8B		
8C		
8D		
8E		
8F		
90		
91		j
92		k
93		l
94		m
95		n
96		o
97		p
98		q
99		r
9A		
9B		
9C		
9D		
9E		
9F		
A0		
A1		
A2		s
A3		t
A4		u
A5		v
A6		w
A7		x
A8		y
A9		z
AA		
AB		
AC		
AD		
AE		
AF		
B0		
B1		
B2		
B3		
B4		
B5		
B6		
B7		
B8		
B9		
BA		
BB		
BC		
BD		
BE		

Hexadecimal Representation	ASCII (7bit)	EBCDIC (8 bit)
BF		
C0		
C1		A
C2		B
C3		C
C4		D
C5		E
C6		F
C7		G
C8		H
C9		I
CA		
CB		
CC		
CD		
CE		
CF		
D0		
D1		J
D2		K
D3		L
D4		M
D5		N
D6		O
D7		P
D8		Q
D9		R
DA		
DB		
DC		
DD		
DE		
DF		
E0		
E1		
E2		S
E3		T
E4		U
E5		V
E6		W
E7		X
E8		Y
E9		Z
EA		
EB		
EC		
ED		
EE		
EF		
F0		0
F1		1
F2		2
F3		3
F4		4
F5		5
F6		6
F7		7
F8		8
F9		9
FA		
FB		
FC		
FD		
FE		
FF		

AN INTRODUCTION
TO MICROCOMPUTERS

BY ADAM OSBORNE

TO ORDER ADDITIONAL COPIES
Adresser toute commande à
Bücher Ausgabe

SYBEX

PUBLICATIONS DEPT.
313 Rue Lecourbe
75015 — PARIS, FRANCE
Telex: 200858 Sybex

SYBEX

AN INTRODUCTION TO MICROCOMPUTERS
VOLUME I — BASIC CONCEPTS

M11